WEST Search History

DATE: Sunday, December 14, 2003

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT,PGPB; PLUR=NO; OP=ADJ			
L11	branch$3 same L10	0	L11
L10	"same block address"	81	L10
L9	(l6 or l7) same branch$3	3	L9
L8	"same bundle address"	0	L8
L7	"same cache line address"	25	L7
L6	"same line address"	64	L6
L5	5805878.uref.	16	L5
L4	l1 same Simultaneous$2	29	L4
L3	l1 same L2	0	L3
L2	"same address"	11952	L2
L1	(multiple branch$2) with predict$3	114	L1

END OF SEARCH HISTORY

Multiple branch and block prediction

Wallace, S. Bagherzadeh, N.
Dept. of Electr. & Comput. Eng., California Univ., Irvine, CA;
This paper appears in: **High-Performance Computer Architecture, 1997., Third International Symposium on**
Meeting Date: 02/01/1997 -02/05/1997
Publication Date: 1-5 Feb 1997
Location: San Antonio, TX , USA
On page(s): 94-103
References Cited: 14
Number of Pages: xi+353
INSPEC Accession Number: 5508103

Abstract:
Accurate branch prediction and instruction fetch prediction of a microprocessor are critical to achieve high performance. For a processor which fetches and executes multiple instructions per cycle, an accurate and high bandwidth instruction fetching mechanism becomes increasingly important to performance. Unfortunately, the relatively small basic block size exhibited in many general-purpose applications severely limits instruction fetching. In order to achieve a high fetching rate for wide-issue superscalars, a scalable method to predict multiple branches per block of sequential instructions is presented. Its accuracy is equivalent to a scalar two-level adaptive prediction. Also, to overcome the limitation imposed by control transfers, a scalable method to predict multiple blocks is presented. As a result, a two black, multiple branch prediction mechanism for a block width of 8 instructions achieves an effective fetching rate of 8 instructions per cycle on the SPEC95 benchmark suite

Index Terms:
computer architecture instruction sets microprocessor chips SPEC95 benchmark suite instruction fetch prediction instruction fetching instruction fetching mechanism microprocessor multiple branch and block prediction

Multiple Branch and Block Prediction

Steven Wallace and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697
swallace@ece.uci.edu, nader@ece.uci.edu

Abstract

Accurate branch prediction and instruction fetch prediction of a microprocessor are critical to achieve high performance. For a processor which fetches and executes multiple instructions per cycle, an accurate and high bandwidth instruction fetching mechanism becomes increasingly important to performance. Unfortunately, the relatively small basic block size exhibited in many general-purpose applications severely limits instruction fetching. In order to achieve a high fetching rate for wide-issue superscalars, a scalable method to predict multiple branches per block of sequential instructions is presented. Its accuracy is equivalent to a scalar two-level adaptive prediction. Also, to overcome the limitation imposed by control transfers, a scalable method to predict multiple blocks is presented. As a result, a two block, multiple branch prediction mechanism for a block width of 8 instructions achieves an effective fetching rate of 8 instructions per cycle on the SPEC95 benchmark suite.

1 Introduction

The goal of a superscalar microprocessor is to execute multiple instructions per cycle. Instruction-level parallelism (ILP) available in programs can be exploited to realize this goal [4]. Unfortunately, this potential parallelism will never be utilized if the instructions are not delivered for decoding and execution at a sufficient rate [9]. A high performance fetching mechanism is required.

Conditional branches create uncertainty in fetching instructions, which can cause severe performance penalties if not accurately predicted. Also, when a control transfer is detected, its target address must be predicted in order to avoid a stall. Even with perfect dynamic branch prediction, predicting one branch per cycle drastically limits performance, due to small basic block sizes. Multiple branch predictions and multiple target addresses need to be predicted in a single cycle in order to overcome this limitation and achieve a high fetching rate [10].

Researchers have shown how to accurately predict conditional branches. Yeh and Patt introduced a two-level adaptive branch prediction. It uses previous branches' history to index into a Pattern History Table (PHT). They report a 97% branch prediction accuracy [14]. Calder and Grunwald proposed the Next Line Set (NLS), which predicts the next instruction cache line and set to fetch [1]. Both the PHT and NLS were designed for a *scalar* processor and only attempt to fetch one instruction per cycle. Yeh also showed how to perform multiple branch prediction using the PHT and a branch address cache [11]. Unfortunately, the cost of this implementation grew exponentially. In this paper, however, we present a *scalable* mechanism to perform multiple branch and multiple block prediction using the PHT and NLS concepts.

Seznec et. al. [8] recently introduced an innovative way to fetch multiple (two) *basic* blocks. Their idea is to always use the current instruction block information to predict the block following the next instruction block. Its accuracy is as good as a single block fetching and requires little additional storage cost. The major drawback, as the authors explain, is that the prediction for the second block is dependent on the prediction from the first block (the tag-matching is serialized). Our scheme, however, is able to predict multiple blocks in parallel without such a dependency.

A *basic* block is defined to be instructions between branches, whether they are taken or not taken. We refer to a block simply as a group of sequential instructions up to a predefined limit, n, or up to the end of a line. Instructions after the first control transfer in a block are not used. A *line* of instructions refers to the group of instructions physically accessed in the instruction cache. The size of a line may be greater than or equal to the block width n.

We first present a method to predict multiple branches in a single block of instructions. Then we present a method to predict the addresses of two blocks in a single cycle. Next, we evaluate the performance of predicting multiple branches and blocks. Finally, we give cost estimates.

0-8186-7764-3/97 $10.00 © 1997 IEEE

2 Multiple Branch Prediction

Yeh and Patt introduced a two-level correlated branch prediction for conditi nal branches [12] capable of predicting one branch per cycle. They also proposed a method for multiple branch prediction [11]. Multiple branches can be predicted in a single cycle by looking up an entry in the PHT using the global history register and also looking up the entries for the two possible outcomes (branch taken or branch not taken) for the first prediction. If three predictions are required, then four additional entries are looked up. This process grows exponentially based on the number of conditional branches predicted.

Although this method retains the accuracy of the original scalar prediction, we have found that this exponential lookup is not necessary and is wasteful. Our solution is a scalable expansion of Yeh's original two-level adaptive branch prediction. All of his schemes involve finding pattern history information to predict a single branch using a 2-bit up/down saturating counter. We expand this pattern history to contain information not for one branch instruction, but for an entire *block* of potential branch instructions. For example, if eight instructions per block are being fetched, a PHT entry will contain eight 2-bit counters, one for each position in a block. One important difference is updating the global history register (GHR) or branch history register (BHR). Instead of being updated after the prediction of each individual branch, it is updated after the prediction for the entire block. For example, if three branches are predicted not taken, not taken, taken, then the GHR/BHR is shifted to the left three bits and a "001" inserted. All of Yeh's original variations may be expanded in this manner, except his per-addr variation now becomes a per-block variation.

Figure 1 is a block diagram of a multiple branch prediction fetching mechanism. While the instruction cache is reading the current block of instructions, the instruction fetch mechanism at a *minimum* must predict the index of the next line to retrieve from the instruction cache. The complete address may be determined later. Therefore, an efficient method to predict target addresses is to use an NLS table. We modify and expand it to be indexed by the instruction *block* address and contain target lines for an entire block of instructions. Alternatively, a Branch Target Buffer (BTB) may be used [6]. The BTB, however, is also modified to be indexed and checked against the instruction block address and contain target addresses for an entire block of instructions. The NLS or BTB may be viewed as n separate tables accessed in parallel, which predict the target address for each of the n possible branch exit positions. The actual target address, if any, is selected at a later time. We call an NLS or BTB which predicts targets for a whole block a *target array*.

In addition, the *branch type* information is no longer contained in the NLS table, but in a separate block instruction type (BIT) table. We have discovered that in superscalar fetch prediction, knowing what type of instructions are in a block is the most critical piece of information. Each BIT entry contains two bits of information for each instruction in a cache line. This BIT information may be pre-decoded and contained in the instruction cache itself. Depending on implementation considerations, a separate array with a faster access time may be required. If a separate table is used, the BIT table may be smaller than the number of lines in the instruction cache at the expense of performance.

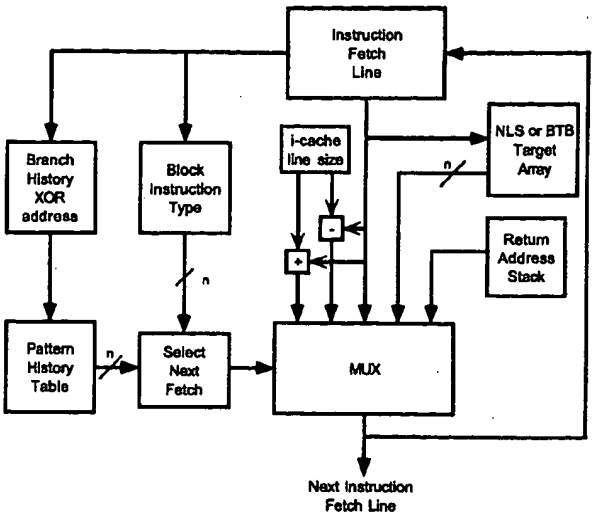


Figure 1. Block diagram of a multiple branch prediction fetching mechanism

At a minimum, the BIT information for each instruction in a fetch block must contain at least two bits to represent that an instruction is either not a branch, a return instruction, a conditional branch, or other types of branches. If we expand this to three bits per instruction, it can contain additional information about conditional branches with targets adjacent to the current line, referred to as *near-block* targets. The offset into the line may be quickly added with a $lg(n)$-bit adder as soon as the branch offset is ready. As a result, near-block target addresses do not need to be stored in the target array, and the size of the target array can be reduced.

Given the starting position in the line fetched, BIT and PHT block information, the instruction fetch control logic uses the instruction type information to find the first unconditional branch or conditional branch predicted to be taken based on its pattern history. The next line to be fetched is then selected from a multiplexer whose input contains the current line, previous line, following line, two lines after the current line, the top of the return address stack (RAS) [5],

and the n possible targets from branches in a block. The BIT types and resulting prediction sources are summarized in Table 1.

Table 1. BIT Types and Predicti n S urces

			Instruction Type	Prediction Source
0	0	0	Non-branch	Fall-through PC
0	0	1	Return	Return Stack
0	1	0	Other branches	Always use Target Array
0	1	1	Conditional branch, long target	Target Array entry or Fall-through, depending on PHT
1	0	0	Cond. branch, prev line	Current line - line size
1	0	1	Cond. branch, same line	Current line
1	1	0	Cond. branch, next line	Current line + line size
1	1	1	Cond. branch, next line+1	Current line + 2 * line size

The processor should keep track of the target address of each conditional branch that is predicted not taken. In the case it was mispredicted, the correct block may be immediately fetched the following cycle after branch resolution. Otherwise, an additional cycle is required to read the target address from the target array.

Table 2 is an example showing a line of instructions and the result of prediction. The type of instruction, BIT information code, and PHT entry values are given. The starting position corresponds to the beginning of a block. The exit position is where an instruction transfers control. For each *possible starting position*, the exit position, next line select prediction, target used for a misprediction, and the new prediction used after a misprediction are shown. NLS(x) indicates that the target address for the exit position x is selected from the NLS target array. For instance, if the starting position is 4, the exit position is 5 where a conditional branch is predicted to be taken and the NLS at position 5 is used for the target address. If the branch is mispredicted, the return address stack is used as the target for the next block. Since the pattern history indicates a "second chance" bit, the prediction will not change the next time the branch is encountered.

3 Multiple Block Prediction

Once an instruction which transfers control is encountered, no more instructions in a block may be used. Another cycle is required to fetch from a different line in the instruction cache. This is a barrier to fetching a large number of instructions in a single cycle. Hence, what is needed is the capability to fetch multiple blocks at the same cycle. The problem is determining which blocks to fetch each cycle.

Fetching two blocks per cycle requires predicting two lines per cycle. In order to accomplish this prediction completely in parallel, *only* the address of the two lines currently being fetched may be used as a basis for prediction. Using the PC from the last block currently being fetched, the first line can be predicted using methods from the previous sections. The difficulty arises in predicting the following (second) line. Yeh and Patt used a branch address cache to give all possible starting basic block addresses based on the current PC [11]. Depending on the branch prediction, the appropriate addresses were selected. The drawback again is the branch address cache grows exponentially with the number of branch predictions.

The underlying problem with predicting two lines to fetch is that the prediction for the second line is *dependent* on the first. Hence, the PHT and BIT information for the second line cannot be fetched until the first line has been predicted and the new PC and GHR have been determined. The solution to this problem is essentially to predict our prediction. The end result of using the BIT and PHT for prediction is a multiplexer selector. Therefore, because the BIT and PHT information for the second block prediction are not available, we store the multiplexer selection bits of a previous prediction for that block into a *select table* (ST). The select table is indexed by the exclusive-or of the GHR and the current PC block address [7]. This index is the same as the index into the PHT for the prediction of the first block. The select value read from the select table is used to directly control the multiplexer for the second block prediction. A 3-bit selector can be used with a block width of four ($n = 4$). Four bits are required for $n = 8$.

3.1 Single Selection

Figure 2 is a block diagram of a dual block prediction fetching mechanism. It has two multiplexors to select the next two lines to fetch. The first selection is calculated from the PHT and BIT information. The second selection comes from the select table. To accurately predict target addresses, a *dual* target array is used. It provides n target addresses for the first target and n target addresses for the second target. The address of the second block currently being fetched is used as the index into both target arrays. Although the NLS must have two target arrays, a BTB may use its tag to indicate the target number (block one or two).

Undesirable duplication of target addresses is inherent to the dual target array. A branch's target address could be stored in both target arrays. Also, it may be represented in the second target array multiple times, since a branch may have multiple predecessor blocks. This duplication, however, does not significantly reduce its accuracy compared to a single target array.

The second multiplexer shown in Figure 2 is dependent on the output of the first multiplexer. An addition to determine the fall-through address of the first prediction or other near-block targets is required. Although the addition of a line index is relatively small, if timing is critical, each of the n targets from the first target array and the RAS can cal-

Table 2. Next line prediction exampl based n starting position

Position in block	0	1	2	3	4	5	6	7
instruction type	shift	branch	add	jump	sub	branch	move	return
BIT value	000	100	000	010	000	011	000	001
PHT value	XX	10	XX	XX	XX	11	XX	XX
exit position	1	1	3	3	5	5	7	7
select prediction	line--	line--	NLS(3)	NLS(3)	NLS(5)	NLS(5)	RAS	RAS
target on misprediction	NLS(3)	NLS(3)	N/A	N/A	RAS	RAS	N/A	N/A
select replacement	NLS(3)	NLS(3)	N/A	N/A	NLS(5)	NLS(5)	N/A	N/A

culate the fall-through (and possibly near target(s)) indexes *before* the first block selector is ready. The fall-through adder used as input for the second multiplexer can now be replaced with a multiplexer which selects the correct pre-computed fall-through address from the first target.

The RAS sends the top of its stack to the input of the first multiplexer. For the second multiplexer, if the first block performs a call, the RAS input is bypassed with the address after the exit address of the first block. If the first block performs a return, the RAS sends the second address off the stack. Otherwise, the top of the stack is sent to the second multiplexer. In addition, the target array should encode whether or not its target is a result of a call, so that proper return bypassing can take place.

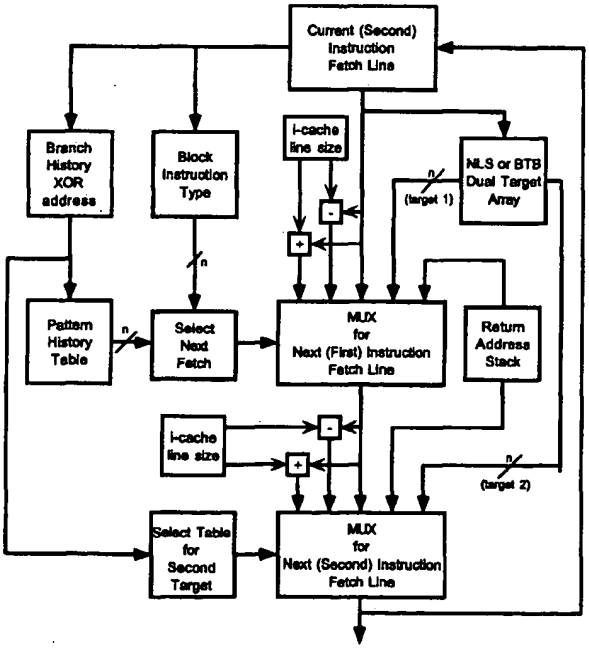


Figure 2. Block diagram for dual block prediction

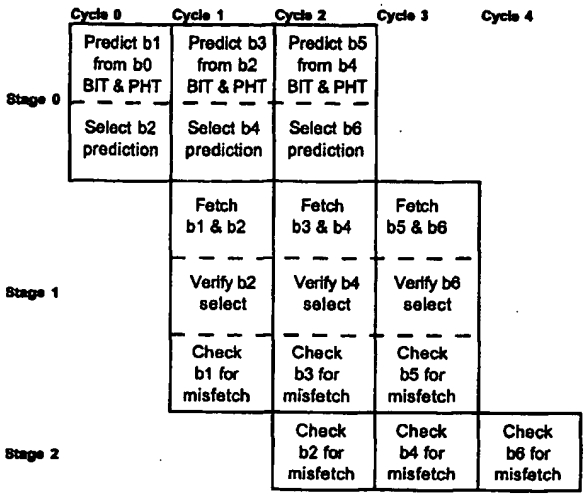


Figure 3. Pipeline stage diagram for dual block prediction

Figure 3 displays the pipeline stages involved in the dual block prediction. The first stage is the prediction of the next two blocks (bX denoted block # X). The selector for the first predicted block is computed from BIT and PHT information. The second block is predicted by reading the select table. The second stage fetches the two blocks. It also verifies the select prediction in the previous stage against the PHT and BIT information which is now available. If the prediction is different, then a *misselect* has occurred. The previous prediction is replaced with the new prediction in the select table, and the new block is fetched. Also during the second stage, the predicted target address of the first block is checked against the calculated branch offset or immediate branch from the previous block (misfetch). The third stage checks for a misfetch of the second block.

From the pipeline diagram, we observe two problems. One problem is with the updating of the GHR. The GHR can reflect the outcome of the first block prediction, but for the second block prediction, there is no information about the number of conditional branches predicted or their outcome. Therefore, a select table entry needs t contain pre-

diction information to update the GHR. This can be accomplished by using $lg(n)$ bits to represent the number of not taken branches and one bit to represent either a fall-through case or a taken branch. The second problem is with near-block select prediction of the second block. It does not give information about the offset into the line. As a result, up to $lg(n)$ extra bits are needed to provide this information, or there may be enough time to calculate the line offset after its source block has been read. Alternatively, one could choose not to use near-block targets to avoid this problem. The GHR and position prediction (if any) are verified at the same time as the select prediction.

3.2 Double Selection

The selection prediction can be used on the first block as well as the second block. We refer to selection prediction of both blocks as *double selection*. Figure 4 is a block diagram of multiple block prediction using double prediction. Double prediction increases the misselect penalty. However, the benefit is the removal of BIT storage altogether. The instruction type is decoded after the line has been fetched. The select table is still indexed by the GHR XOR starting address, but it is now a *dual* select table, providing selectors for both multiplexors. Timing concerns regarding the calculation of the selector for the first target no longer exist. The potential for timing problems from the adders between the multiplexers is significantly reduced. Selector and GHR prediction bits for both blocks are required, although the starting position prediction for the second block is no longer needed.

Figure 5 is a pipeline diagram using double selection. The first stage predicts the next two blocks from the dual select table. The second stage fetches the two blocks, and verifies the first block's select prediction and target address. The third stage verifies the second block's select prediction and target address.

3.3 Misprediction

The penalties for the different types of possible mispredictions are listed in Table 3. It is assumed that it takes four cycles to resolve a branch after it has been fetched. For the first block, if there are remaining instructions required to be re-fetched after a conditional branch was mispredicted taken, then it will take an additional cycle. A misprediction on the second block always requires another cycle. There is a one cycle misselect or GHR mispredict penalty using a single select on the second block.

With a double selection prediction, the first block has a one cycle penalty while the second block takes two cycles. Since a misselect is detected during or immediately after the instructions have been fetched, instructions that would

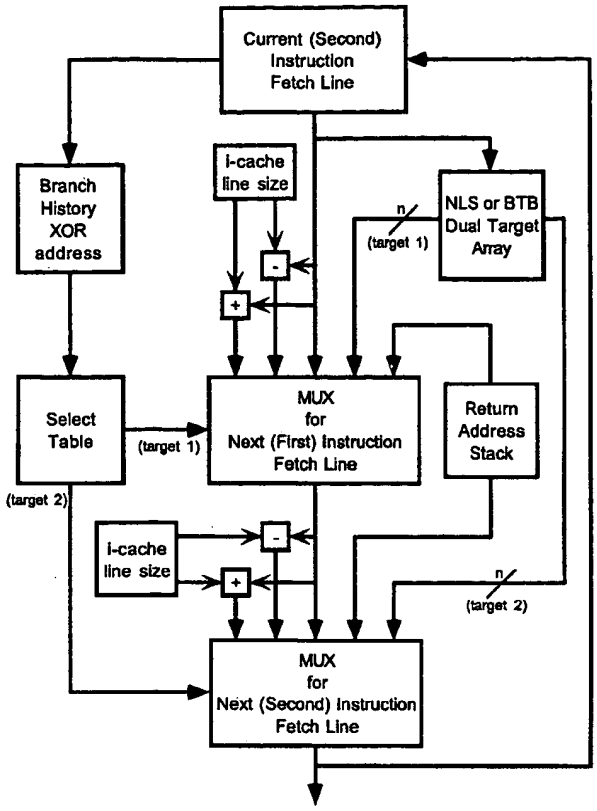


Figure 4. Block diagram for dual block prediction using double selection

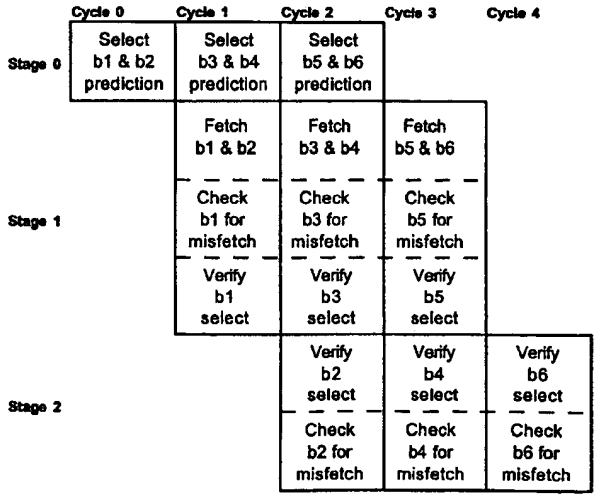


Figure 5. Pipeline stage diagram for dual block prediction using double selection

have been discarded on a taken branch become valid, and no re-fetch cycle is needed. A misfetch takes one cycle for the first block and two cycles for the second block to detect.

Since multiple blocks are being fetched using different cache lines, a multiple banked instruction cache is required. Since two lines are fetched simultaneously, they may map into the same cache bank. Should a conflict arise, the second line is read the next cycle.

Table 3. Misprediction Penalties

Misprediction	Single Select		Double Select	
	1^{st} blk	2^{nd} blk	1^{st} blk	2^{nd} blk
Conditional branch	4*	5	4*	5
Return	4	5	4	5
Misfetch indirect	4	5	4	5
Misfetch immediate	1	2	1	2
Misselect	N/A	1	1	2
GHR	N/A	1	1	2
BIT	1	1	N/A	N/A
I-cache bank conflict	0	1	0	1

* Add one cycle if instructions remain and need to be re-fetched.

In order to facilitate recovery from a mispredicted branch, each conditional branch is assigned a bad branch recovery (BBR) entry, which provides information on how to update branch prediction tables and provide a new target. The processor must create this entry and keep track of it as the branch moves down the pipeline. Table 4 lists a description and sizes of the fields in a recovery entry. A recovery entry is created when the block which contains a conditional branch is predicted using BIT and PHT information. When a prediction is made for a conditional branch, another prediction is made for that block assuming its original prediction is incorrect. If a branch is predicted not taken, then the alternate target address is the branch's target address. If it is predicted taken, then the alternate address is the next control transfer or fall-through address in its block (see the example in Table 2). The alternate target address is entered into the recovery entry. In addition, a replacement selector and new GHR are generated.

Table 4. Bad Branch Recovery Entry

Bits	Description
1	Block 1 or 2
1	Predicted taken or not taken.
1	Second chance.
8-12	PHT index.
2n	PHT block (optional).
8-12	Corrected GHR.
8-11	Replacement selector.
10/30	Corrected i-cache index or full addr.

The recovery entry may also contain the entire PHT block that reflects a successful pattern history update for each branch in the block up to the current branch. After the last branch in a block has been successfully resolved, it uses this field to update the PHT to reflect a *correct* prediction. When a branch is mispredicted, it is modified to reflect an *incorrect* prediction (by using the original second chance information) and written to the PHT. If the PHT is not updated by using a PHT block field, then the pattern history for a branch has to be updated using a read/modify/write cycle to the PHT block for each individual branch when it is resolved.

If the branch does not have a "second chance" when it is mispredicted, then the pre-computed selector from the bad branch recovery entry is written into the select table.

If a misprediction occurs for the second block, then any remaining instructions from the first block are fetched along with a new second block target retrieved from the recovery entry. On the other hand, if the misprediction occurs for the first block, an extra cycle may be required to fetch any remaining instructions from the previous block.

4 Performance

The performance of different variations of multiple branch and block prediction was determined by running the SPEC95 benchmark suite on the SPARC architecture. The suite was compiled using the SunPro compiler with standard optimizations (-O). Programs were simulated using the Shade instruction-set simulator [2]. Each program ran for the first one billion instructions.

All the results presented use a block width of eight ($n = 8$). Single selection is used for dual block prediction unless otherwise noted. The results presented only use a global adaptive branch prediction scheme using one global blocked pattern history table. The default size of a select table is 1024 entries, which corresponds to a GHR length of 10 bits. The size of the RAS is 32 entries. It was assumed the processor would always have enough bad branch recovery entries available. Instruction cache misses were not simulated, i.e., a perfect instruction cache was assumed. The only consideration for the instruction cache were the line size and bank conflicts. A line size equal to the block width was used, and the instruction cache was split into eight banks. Also, by default, near-block prediction is not used.

The default target array is a 256-entry NLS array. The set prediction was not simulated. Therefore, the results presented for the NLS configuration are really a direct-mapped tag-less BTB. The performance of a real NLS is affected by the instruction cache configuration. For a performance and cost comparison of an NLS verses a BTB, please refer to [1].

We compare the performance of different types of multiple branch and block architectures using two metrics, as used by Yeh and Patt [13]. The first is the branch execution

penalty,

$$BEP = \frac{Total\ Penalty\ Cycles}{\#Branches}$$

If we assume the other parts of the processor are ideal, we can compute the effective instruction fetch rate,

$$IPC\_f = \#Valid\ instructions / \#Fetch\ cycles$$

where the number of fetch cycles is equal to

$$\#Total\ Penalty\ Cycles + \frac{\#Blocks\ fetched}{Maximum\ blocks\ per\ cycle}$$

The BEP gives information regarding performance and the interaction between the many different types of penalties as listed in Table 3. Nevertheless, all the types of penalties are recorded. Overall performance is best understood from the effective instruction fetch rate. One cannot directly compare a scalar BEP with a superscalar BEP or a multi-block BEP since higher penalties are overcome by increased number of instructions per successful fetch block.

Also, when fetching two blocks per cycle of potentially eight instructions each, up to sixteen instructions may be returned in one cycle. Consequently, the effective instruction fetching rate, IPC_f can be greater than n. If an eight issue processor is used, then extra instructions returned can be buffered [10]. When the raw two block rate is greater than n, the issue unit will usually receive, and average close to, n instructions per request. Of course, a simpler configuration to satisfy issue unit constraints in such a situation would be to use two blocks of four instructions each. This would still yield an excellent fetching rate.

4.1 Conditional Branch Accuracy

To begin with, the conditional branch accuracy of a blocked PHT for multiple branch prediction was evaluated. The branch history length varied from 6 to 12, and the results were compared to a scalar PHT. The scalar scheme used a per-addr PHT with 8 PHTs to give it equal size of a blocked PHT for $n = 8$. Figure 6 displays the branch misprediction rates (*line*) and the improvement over a scalar PHT (*bar*). The difference in accuracy between the scalar and blocked schemes across all variations were small, and the accuracy favored the blocked PHT scheme for most programs. The accuracy of SPECint95 averaged 91.5% while the accuracy of the SPECfp95 averaged 97.3%, using a GHR length of 10. In this case, the blocked PHT had a better accuracy by a few hundredths of a percent for SPECfp95 and a few tenths of a percent for SPECint95.

4.2 BIT

Correct instruction type information for a block is critical to making accurate predictions. Incorrect BIT information can still result in a correct prediction, but this possibility is reduced with larger block sizes. Different BIT table sizes were simulated to evaluate its impact. Using single block fetching, Figure 7 shows the BEP contribution from inaccurate BIT information (*bar*). Also shown is the IPC_f (*line*). Small sized BIT tables result in poor performance. Only until about 2048 entries does the percentage of BEP drop below 5%. Therefore, for smaller sized instruction caches, it may be more beneficial to store the BIT information inside the instruction cache. Conversely, a separate BIT table would be more cost effective because the one cycle miss penalty of the BIT is much lower than an instruction cache miss. The rest of the results presented use two blocks and assume BIT information is stored in the instruction cache.

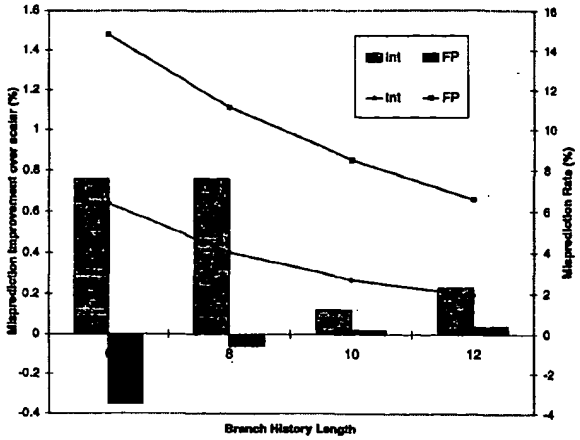
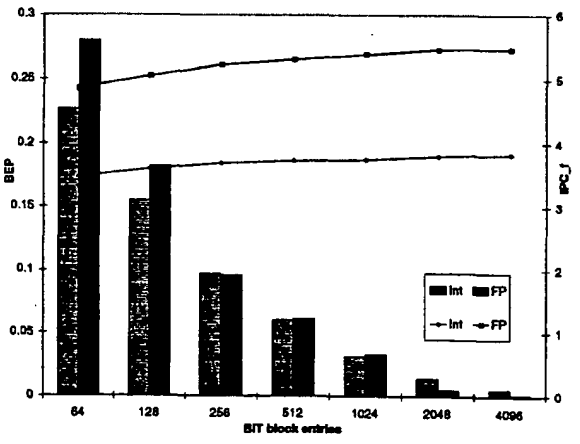


Figure 6. Branch Misprediction Rate and Improvement



Figur 7. BIT P nalty and P rf rmanc

4.3 Single vs. Double Selection

The performance of the select table depends on the branch history length and the number of select tables used. Multiple select tables are indexed by the starting position from the current address. The correct target depends on the entering position in a block, so multiple select tables help identify which target should be selected. The least significant bits of the starting address determine which select table is used. Figure 8 shows the performance of dual block prediction for single and double selection. The global history register length varies from 9 to 12. There can be 1, 2, 4, or 8 STs. However, there are not multiple PHTs. The results demonstrate that increasing the number of STs improves performance as well as increasing the branch history length. The extra penalties from using double selection significantly reduced performance, roughly 10% for most cases. Hence, single selection should be used if implementation considerations permit. Double selection significantly improves, though, with more STs.

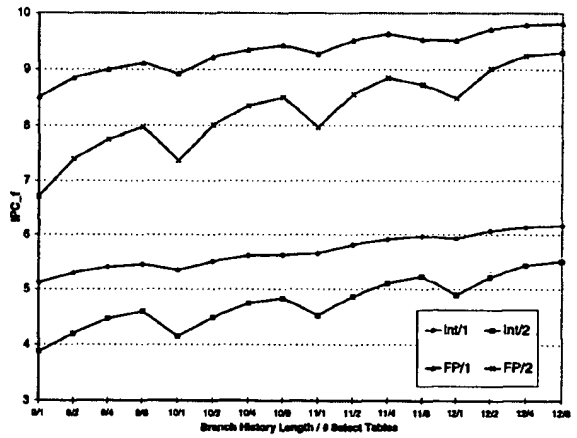


Figure 8. Integer and Floating Point performance using Single and Double Selection

4.4 Target Arrays

Target arrays can use a BTB or NLS. In addition, if a near-block target is used, this will reduce the number of immediate targets used in the target array. Table 5 shows the percentage of BEP due to indirect and immediate misfetches for SPECint95. The total BEP and IPC_f are also reported. The number of *block* entries is varied for both NLS and a 4-way BTB using LRU replacement algorithm. A BTB entry can be for the first or second target, whil an NLS entry has two separate targets. The data indicates that eight NLS block entries are needed for comparable performance of one 4-way BTB entry. About 70% of the conditional branches are near-block targets. As a result of using near-block encoding, the number of BTB or NLS entries can be reduced in half for about the same performance.

Table 5. Indirect and Immediate Misfetch Penalty Comparison for Different Target Array Configurations

Target Type	# blk entries	near-block?	%BEP misfetch imm.	%BEP misfetch ind.	BEP	IPC_f
BTB	8	no	19.2	18.7	0.603	5.02
BTB	8	yes	10.6	16.3	0.520	5.40
BTB	16	no	12.6	15.1	0.523	5.32
BTB	16	yes	6.5	12.6	0.476	5.57
BTB	32	no	7.4	11.6	0.473	5.58
BTB	32	yes	3.6	9.6	0.446	5.73
BTB	64	no	4.0	9.6	0.447	5.72
BTB	64	yes	1.9	7.9	0.431	5.80
NLS	64	no	12.0	14.7	0.516	5.41
NLS	64	yes	6.7	13.1	0.480	5.54
NLS	128	no	8.3	12.3	0.481	5.53
NLS	128	yes	4.2	10.8	0.454	5.67
NLS	256	no	5.5	10.1	0.457	5.66
NLS	256	yes	2.7	8.7	0.438	5.77
NLS	512	no	3.8	9.2	0.444	5.74
NLS	512	yes	1.6	7.9	0.429	5.81

4.5 Instruction Cache Configurations

The performance can be dramatically improved if a different type of instruction cache configuration is used. Using the same line size and block width of eight instructions, the number of valid instructions in a block has been limited due to misalignment. The target address of a control transfer can be in the middle of a line, thus reducing the size of a block. To increase the number of instructions per block (IPB), the cache line size can be extended to 16 instructions [10]. Only up to 8 instructions are returned as a block, but the probability less than 8 instructions are found has been reduced. To solve this problem completely, a self-aligned cache can combine two consecutive lines to form a block [3, 10]. If a self-aligned cache is used though, the number of banks should be doubled to offset the increase in bank conflicts, since up to four lines are being simultaneously accessed to return two blocks. Although there are no bank conflicts with single block fetching, the extended and self-aligned caches improve the instructions fetched per block (IFB) and overall fetching performance.

With the extended and self-aligned caches, when branch prediction is performed using the PHTs, the values wrap around the PHT block. Also the target arrays must be correspondingly extended or self-aligned. The performance of these three cache types are compared using one and two block fetching with single selection. The results are shown

in Table 6, using 8 STs and a branch history length of 10. Outstandingly, the self-aligned cache achieves 10.9 IPC_f for the floating point benchmarks. It averages over 8 IPC_f for the entire SPEC95 suite. The high performance is primarily due to the increase in IFB. Also, the starting address becomes more random which helps associate a select table and use it efficiently. The performance of the extended cache type is between a normal and self-aligned cache. Compared to single block prediction, dual block prediction results in an effective fetching rate approximately 40% higher for integer programs and 70% higher for floating point programs.

Table 6. Instructions per block (IPB) and IPC_f for different cache types

cache type	line size	bnks	SPECint95 IPB	SPECint95 IPC_f 1 blk	SPECint95 IPC_f 2 blk	SPECfp95 IPB	SPECfp95 IPC_f 1 blk	SPECfp95 IPC_f 2 blk
normal	8	8	5.01	3.96	5.66	5.81	5.48	9.43
extend	16	8	5.30	4.12	5.87	6.03	5.65	9.80
align	8	16	5.99	4.53	6.42	6.76	6.33	10.88

Using a self-aligned cache, 8 STs, and a branch history length of 10, Figure 9 shows the BEP of each program and the contribution of BEP by each type of misprediction as described in Section 3.3. The effective instruction fetching rate is inversely proportional to BEP. The most significant BEP contribution is from misprediction of conditional branches. Misselection is the next most significant contribution. Target array mispredictions are also a significant factor in BEP. Some of the floating point programs performed exceedingly well. On the other hand, some integer programs had a high BEP because of poor conditional branch prediction.

5 Cost Estimates

Table 7. Simplified hardware cost estimates

Symbol	Description
n	block width
k	history register length
p	number of PHTs
s	number of Select Tables
t	number of NLS block entries
l	size of line index
a	cache associativity
b	number of BBR entries
i	number of BIT block entries

Table	Simple hardware cost estimate
PHT	$p \times 2^k \times n \times 2$
ST	$s \times 2^k \times 2 \times (lg(n)+1)$
NLS	$t \times n \times (l + lg(a))$
BIT	$i \times n \times 2$
BBR	$b \times (2k + l + lg(a) + 2lg(n) + 5)$

Table 7 lists a simplified hardware cost estimates for the PHT, ST, NLS, and BIT tables. If we use a block width of 8, a 32 KByte direct-mapped instruction cache, a 10-bit history register, 1 PHT, 1 ST, 256 NLS entries, 1024 BIT entries, and 8 BBR entries, the cost estimates evaluate to:

- **PHT:** 16 Kbits
- **ST:** 8 Kbits
- **NLS:** 20 Kbits
- **BIT:** 16 Kbits
- **BBR:** .3 Kbits
- **single block total:** 52 Kbits
- **dual block, single select total:** 80 Kbits
- **dual block, double select total:** 72 Kbits

As the number of instructions that can be predicted in a block increase, the cost increases proportionally. In addition, it is possible to predict more than two blocks per cycle. In that case, the cost grows proportionally to the number of blocks predicted. Another block prediction basically requires another select table and target array, and another read/write port to the PHT and BIT tables.

6 Conclusion

A scalable mechanism to predict multiple branches in a single block was presented. Its conditional branch accuracy is essentially the same as a scalar two-level adaptive branch prediction of equal size. By recording previous predictions in a select table, two blocks can be fetched in a single cycle. Dual block prediction uses either a single or double select table. Double selection may be used at the expense of a slower fetching rate, but may be desirable in processors with deep pipelines and extremely fast cycle times. An extremely high rate can be achieved with a multiple branch and block prediction mechanism.

References

[1] B. Calder and D. Grunwald. Next cache line and set prediction. In *22nd Annual International Symposium on Computer Architecture*, pages 287–296, June 1995.

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS*, 1994.

[3] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[4] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.

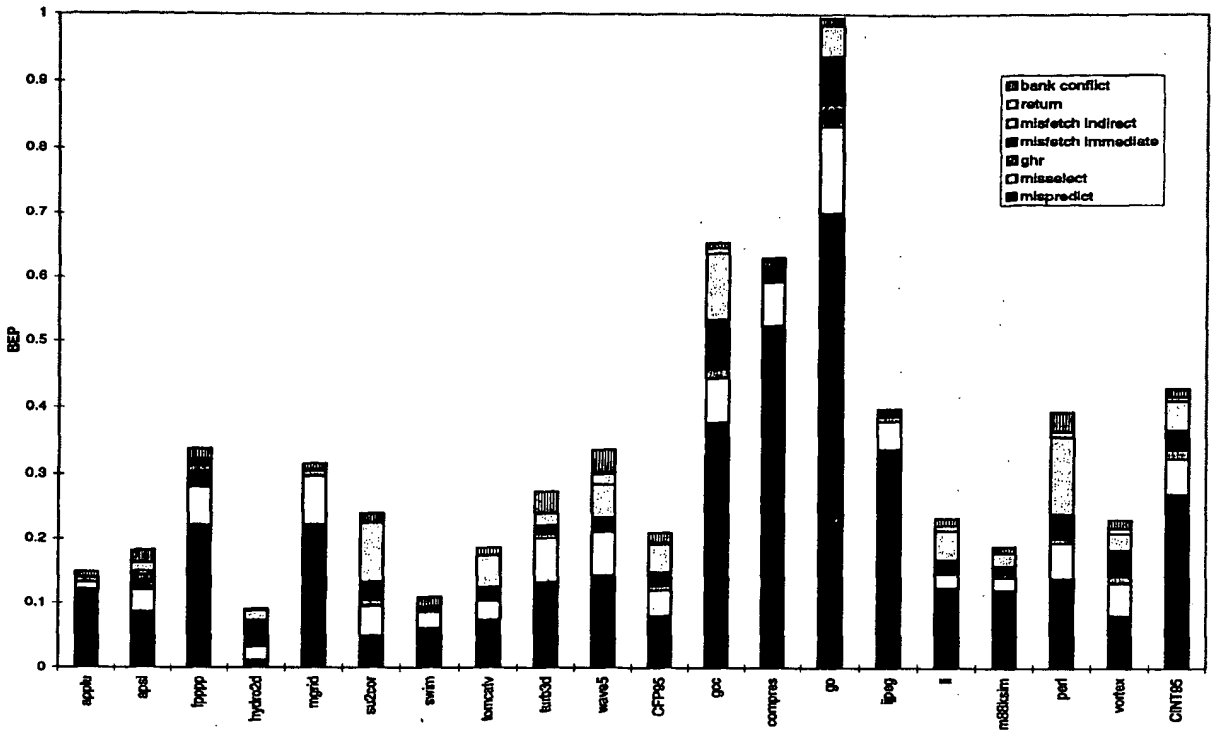


Figure 9. Branch Execution Penalties for two block, single selection

[5] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual International Symposium on Computer Architecture*, pages 34–42, May 1991.

[6] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, 1984.

[7] S. McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[8] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1996.

[9] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 290–302, April 1989.

[10] S. Wallace and N. Bagherzadeh. Instruction fetching mechanisms for superscalar microprocessors. In *Euro-Par '96*, August 1996.

[11] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *7th ACM International Conference on Supercomputing*, pages 67–76, Tokyo, Japan, July 1993.

[12] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, pages 124–134, Gold Cost, Australia, May 1992.

[13] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th Annual ACM/IEEE International Symposium on Computer Microarchitecture*, pages 129–139, Dec. 1992.

[14] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, California, May 1993.

Optimization of instruction fetch mechanisms for high issue rates

Conte, T.M. Menezes, K.N. Mills, P.M. Patel, B.A.
Dept. of Electr. & Comput. Eng., South Carolina Univ., Columbia, SC;

This paper appears in: **Computer Architecture, 1995. Pr ceedings. 22nd Annual International Symposium on**
Meeting Date: 06/22/1995 -06/24/1995
Publication Date: 22-24 Jun 1995
Location: Santa Margherita Ligure , Italy
On page(s): 333-344
References Cited: 21
INSPEC Accession Number: 5086798

Abstract:
Recent superscalar processors issue four instructions per cycle. These processors are also powered by highly-parallel superscalar cores. The potential performance can only be exploited when fed by high instruction bandwidth. This task is the responsibility of the instruction fetch unit. Accurate branch prediction and low I-cache miss ratios are essential for the efficient operation of the fetch unit. Several studies on cache design and branch prediction address this problem. However, these techniques are not sufficient. Even in the presence of efficient cache designs and branch prediction, the fetch unit must continuously extract multiple, non-sequential instructions from the instruction cache, realign these in the proper order, and supply them to the decoder. This paper explores solutions to this problem and presents several schemes with varying degrees of performance and cost. The most-general scheme, the collapsing buffer, achieves near-perfect performance and consistently aligns instructions in excess of 90% of the time, over a wide range of issue rates. The performance boost provided by compiler optimization techniques is also investigated. Results show that compiler optimization can significantly enhance performance across all schemes. The collapsing buffer supplemented by compiler techniques remains the best-performing mechanism. The paper closes with recommendations and suggestions for future

Index Terms:
I-cache miss ratios branch prediction cache design cache storage collapsing buffer compiler optimization techniques computer architecture high issue rates highly-parallel superscalar cores instruction fetch mechanisms optimisation performance performance evaluation superscalar processors I-cache miss ratios branch prediction cache design cache storage collapsing buffer compiler optimization techniques computer architecture high issue rates highly-parallel superscalar cores instruction fetch mechanisms optimisation performance performance evaluation superscalar processors

Documents that cite this d cument

Select link to view other documents in the database that cite this one.

Optimization of Instruction Fetch Mechanisms for High Issue Rates

Thomas M. Conte Kishore N. Menezes Patrick M. Mills Burzin A. Patel
Computer Architecture Research Laboratory
Department of Electrical and Computer Engineering
University of South Carolina
Columbia, South Carolina

Abstract

Recent superscalar processors issue four instructions per cycle. These processors are also powered by highly-parallel superscalar cores. The potential performance can only be exploited when fed by high instruction bandwidth. This task is the responsibility of the instruction fetch unit. Accurate branch prediction and low I-cache miss ratios are essential for the efficient operation of the fetch unit. Several studies on cache design and branch prediction address this problem. However, these techniques are not sufficient. Even in the presence of efficient cache designs and branch prediction, the fetch unit must continuously extract multiple, non-sequential instructions from the instruction cache, realign these in the proper order, and supply them to the decoder. This paper explores solutions to this problem and presents several schemes with varying degrees of performance and cost. The most-general scheme, the collapsing buffer, *achieves near-perfect performance and consistently aligns instructions in excess of 90% of the time, over a wide range of issue rates. The performance boost provided by compiler optimization techniques is also investigated. Results show that compiler optimization can significantly enhance performance across all schemes. The* collapsing buffer *supplemented by compiler techniques remains the best-performing mechanism. The paper closes with recommendations and suggestions for future.*

ISCA '95, Santa Margherita Ligure Italy

1 Introduction

The recent MIPS R10000, Sun UltraSPARC and AMD K5 superscalar processors issue four instructions per cycle, with higher issue rates expected [1],[2],[3]. These processor designs employ multiple functional units and aggressive hardware scheduling to extract parallelism in the instruction stream. Next generation superscalar processors will most likely employ multithreading to further enhance parallelism. These highly parallel execution cores must be fed by sufficient instruction bandwidth, requiring optimized fetch unit design.

Fetching of instructions is constrained by three major factors: instruction cache performance, taken or indirect branches in the fetch stream, and instruction alignment. The design of the instruction cache has received much attention [4],[5],[6]. This body of work includes compiler techniques to enhance instruction cache performance [4],[7],[8]. The combined effect of this work is to lessen the impact of instruction cache misses on fetch bandwidth. Branch prediction is the second factor that constrains fetching. Several recent studies address the accuracy of branch prediction [9],[10],[11]. But branch prediction alone is not sufficient to deliver high fetch bandwidth. Even when branches are predicted accurately, the fetch unit must extract multiple, non-sequential instructions from the instruction cache in one cycle. The layout of instructions in the cache often frustrates this task. For high instruction bandwidth at high issue rates, the fetch unit must realign instructions in the predicted order, then pass the instructions on to the decode and execution units. Thus the third constraint on instruction fetch is due to the alignment of instructions in cache blocks. This problem is just emerging as issue rates increase beyond two instructions per cycle. This paper develops several solutions to the alignment problem.

Several approaches to high-bandwidth instruction

fetch have been implemented for commercial processors. Most decouple the instruction fetch unit from the execution unit via queues, and allow the fetch unit to speculate beyond branches [1],[12]. This decoupling reduces the impact of more-complicated (and higher-latency) instruction fetch hardware. In addition to this, the six instruction per cycle IBM POWER2 architecture employs an instruction cache with eight, independently-addressable banks [13]. This fetch unit can align many instruction sequences, but is limited by the POWER2's static branch prediction mechanism, which is known to have lower performance than dynamic schemes. The recently-announced AMD superscalar 29K addresses this limitation by embedding prediction and branch target address information in the cache array to enable a taken branch to be resolved without penalty [3]. However, this scheme cannot handle short branches within a cache block (e.g., hammocks), or multiple branches in one fetch, both of which are encountered frequently for integer code.

This paper presents several schemes of increasing complexity that address the instruction alignment problem. Implementation details are discussed for all the schemes. All comparisons are based on simulated results of the IPC for three microarchitectures. The results show that the most-complex scheme, the *collapsing buffer*, efficiently handles short forward branches and many cases of multiple branches. It achieves performance near the theoretical upper bound for a highly-parallel, 12 instruction issue microarchitecture. The effects of compiler optimizations on the performance of the schemes is also studied. The profile-driven code reordering optimization is found to be highly successful, significantly enhancing the performance of all schemes. A second optimization, *nop* insertion for branch target alignment, produces mixed results, suggesting this optimization plays only a secondary effect. The data is used to suggest several approaches for instruction fetch design at high issue rates.

The remainder of this paper is organized into three sections. The following section presents the machine model, the experimental technique, and other related assumptions employed in this study. This is followed by a discussion of the lower and upper bounds for instruction alignment performance. These bounds are termed *sequential* and *perfect* alignment, respectively. The designs and performance of the proposed hardware schemes are then discussed. The effect of compiler optimization is analyzed to find a balance between hardware and software solutions. The paper closes with recommendations and suggestions for future work in this area.

2 Experimental setup

The results that follow are presented for all six SPECint92 benchmarks, three additional integer benchmarks (`mpeg_play`, `bison`, and `flex`), and six SPECfp92 benchmarks. The benchmarks were compiled using GCC with the compiler options "`-O -fschedule-insn`." The latter option invokes a dag-based local scheduler. Experiments with this option show that it marginally enhances parallelism. All experiments were run using HP 9000/735-class workstations. The instruction set used for pipeline simulation is a simplified version of GCC's intermediate code captured after PA-RISC-specific register allocation but before final code generation. Instructions are encoded using a fixed, 32-bit format.

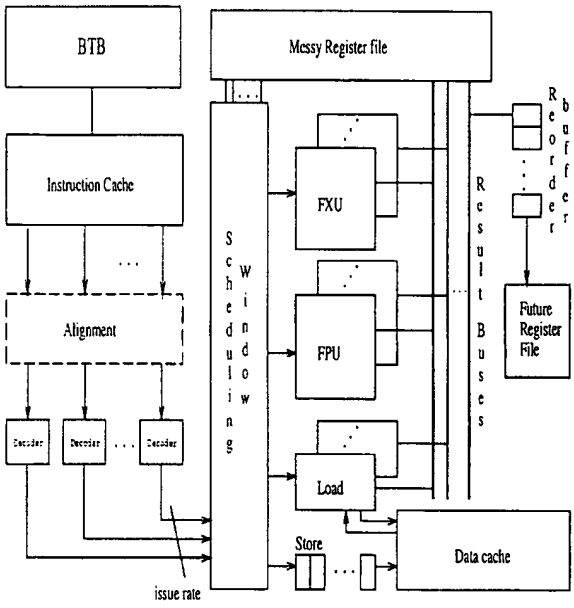


Figure 1: Structure of simulated microarchitecture.

Traces were captured using the *spike* tracing tool and then fed into a processor simulation. This simulation assumes a full-Tomasulo, out-of-order execution microarchitecture, depicted in Figure 1. Three versions of the microarchitecture are discussed in this paper, and their parameters are summarized in Table 1. All three versions have a scheduling window that resolves dependencies and implements Tomasulo-style renaming via tags. Entries in this window correspond to generic reservation stations. This window also serves to decouple the fetch unit from the execu-

tion unit, allowing the fetch unit to speculate ahead in the instruction stream. Speculative execution of more than one predicted conditional branch is supported via the precise interrupt facility (see below). The three classes of microarchitectures support differing degrees of speculation, in proportion to their issue rates. For example, the PI4 microarchitecture issues four instructions per cycle. Experiments with the degree of speculation showed that speculative execution beyond two branches was required to keep the pipeline full. Similarly, the PI8 architecture supports speculation beyond four, and the PI12 supports speculation beyond six branches.

Independent instructions are fired from the window into the execution core, which is composed of fixed-point units (FXUs), floating-point units (FPUs), branch units, and the data cache interface. Access to the data cache is through load units and a store buffer. Data cache misses are not explicitly modeled in the simulator. The PI4 model has two fixed-point units (FXU's), two floating-point units (FPU's), and two branch units. The PI8 model is similar, but scaled by doubling its resources to create a more parallel microarchitecture. The issue rate is increased to eight instructions per cycle. The PI12 model follows this design pattern, with an issue rate of 12 instructions per cycle.

Completing instructions are distributed via result buses. The number of result buses equals the total number of function units, so that bus contention seldom occurs. Two register files are maintained: the *Messy register file* and the *Future register file*. The former is used for out-of-order execution. If used without augmentation, the microarchitecture would be limited to imprecise interrupts. This is remedied using a reorder buffer [13]. The chief performance metric is *instructions retired per cycle* (IPC), which is the number of instructions leaving the reorder buffer (i.e., retiring) per simulated execution cycle.

All three microarchitectures have direct-mapped instruction caches. The cache block size is calculated so that a block holds the maximum issue rate of instructions. PI4 has size 16B, PI8 has size 32B, and PI12 has size 64B blocks. The cache sizes are also scaled with issue rate: 32KB (PI4), 64KB (PI8), and 128KB (PI12).

A branch-target buffer employing a 2-bit counter predictor is used for this study. The buffer is direct-mapped and has 1024 entries, comparable to commercial BTB designs (e.g., 512 entries for the Pentium [14], or 256/512 entries for the decoupled PowerPC 604 BTB [15]). Branch target addresses are also cached in the BTB for each entry. The BTB is interleaved into multiple banks with an interleave factor equal to the number of instructions in a cache block (e.g., an interleave factor of 4 for PI4). BTB interleaving is discussed further below.

Table 1: Machine model parameters: PI4, PI8, and PI12.

PI4 Machine model	
Issue rate	**4** instructions/cycle
Window queue	**16** entries
Instruction cache	32KB, dir. mapped, 16B blocks
Fixed-point unit	**2**, with latency = **1** cycle
Floating-point unit	**2**, with latency = **2** cycles
Branch unit	**2**, with latency = **1** cycle
Speculation	Speculates beyond **2** branches
PI8 Machine model	
Issue rate	**8** instructions/cycle
Window queue	**24** entries
Instruction cache	64KB, dir. mapped, 32B blocks
Fixed-point unit	**4**, with latency = **1** cycle
Floating-point unit	**4**, with latency = **2** cycles
Branch unit	**4**, with latency = **1** cycle
Speculation	Speculates beyond **4** branches
PI12 Machine model	
Issue rate	**12** instructions/cycle
Window queue	**32** entries
Instruction cache	128KB, dir. mapped, 64B blocks
Fixed-point unit	**6**, with latency = **1** cycle
Floating-point unit	**6**, with latency = **2** cycles
Branch unit	**6**, with latency = **1** cycle
Speculation	Speculates beyond **6** branches
Parameters common to all machine models	
Interlocking	Full Tomasulo, out-of-order
Branch target buffer	1024-entry buffer, 2-bit counter

3 Hardware Fetch Mechanisms

The lower bound for instruction fetch bandwidth is one instruction per cycle in the presence of a cache hit and a correctly predicted branch. However, few fetch mechanisms perform so poorly. A more-realistic lower bound is the performance of a sequential block fetch scheme. Such a scheme fetches an entire cache block and then selects multiple instructions from the block. This removes the normal cache word select logic and replaces it with masking logic. No hardware is provided to handle short branches inside the block (*intra-block branches*). The only code sequences that are handled by the technique are sequential instructions. For this reason, the technique will be called *sequential* throughout this paper. The operation of *sequential* is depicted in Figure 2 for a short program fragment.

The upper bound of instruction fetch bandwidth is when the pipeline is never starved due to a lack

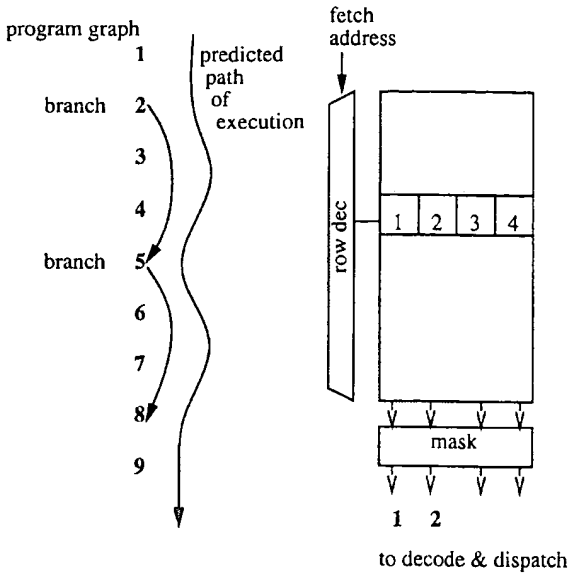


Figure 2: Example operation of *sequential* for sequence **1**, **2**, **5**, **8**.

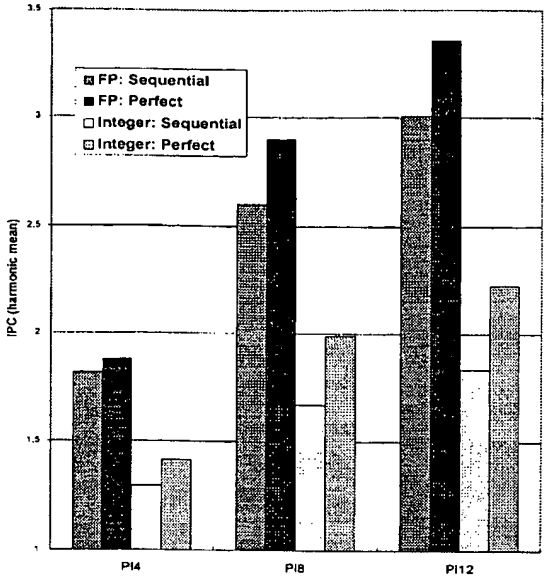


Figure 3: Performance of *sequential* versus *perfect* for integer and floating-point benchmarks.

of instructions. This bound is referred to as *perfect*. Specifically, *perfect* assumes that the instruction memory bandwidth into the scheduling window is unlimited (in the absence of instruction cache misses). Figure 3 presents the harmonic mean of the IPC for *sequential* and *perfect* for the integer and floating-point benchmarks. The data justifies the need for better instruction fetching for all machines, with the possible exception of floating-point code executing on the PI4 architecture. The loop-intensive floating-point benchmarks exhibit regular access patterns, reducing the need for better fetch mechanisms. The integer benchmarks require more effective mechanisms for better performance, due to a higher dynamic frequency of branch instructions.

3.1 Interleaved sequential

One enhancement to *sequential* is to interleave the instruction cache into two banks and prefetch one sequential block in advance. This *interleaved sequential* scheme (Figure 4) achieves higher effective issue rates over plain *sequential* for accesses that span block boundaries. Non-sequential accesses are not allowed. For example, if the sequence were **1**, **2**, **5**, **8**, as in Figure 2, the hardware would not be able to remove the useless instructions between **2** and **5**. As another example, assume *interleaved sequential* is fetching instructions in cache blocks A, B, C, D, etc. The cache is accessed for A and B, then B and C, then C and D, etc.

The *interleaved sequential* scheme must determine and eliminate any predicted non-sequential instructions before forwarding to the decoder. This is accomplished using a BTB interleaved by the number of instructions in a cache block [9]. A BTB query returns the successor block address and a bit-pattern predicting which instructions in the fetched block are valid for decoding. The successor block address is used to invalidate the sequential prefetch block. The block address and bit-pattern are found using a chain of comparators (depicted in Figure 5). Delay through the chain is proportional to the number of instructions in a cache block times the comparator propagation delay. (If this is significant, the chain can be redesigned using generate/propagate logic to reduce the delay.)

Two additional hardware entities are included to assist instruction aligning. These are the *interchange switch* and the *valid select* logic. The *interchange switch* can reverse the order of the fetch block and the target block. For example, if the fetch block is in the right-hand bank in Figure 4 and the target

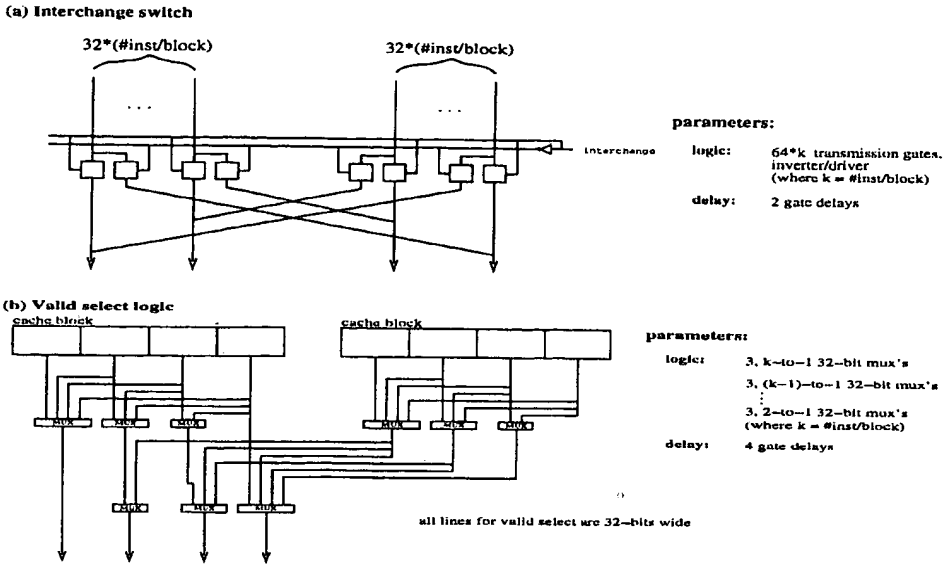


Figure 6: Design details of (a) the interchange switch, and (b) the valid select logic for *interleaved sequential* and *banked sequential.*

block is in the left-hand bank, the two blocks must be reordered so that instructions fed to the decoder are sequential. The design of the interchange switch that performs this task is shown in Figure 6(a). This design requires $64 \times k$ transmission gates for cache blocks that hold k, 32-bit instructions per block.

The *valid select* logic has the responsibility of selecting the valid instructions from the two cache blocks. For an input of $2 \times k$ instructions, this logic selects the first k sequential, valid instructions as determined by the BTB prediction information. It requires an array of 32-bit multiplexers, and has nominal delay. The design of *valid select* is shown in Figure 6(b) (the right-most multiplexer is only required for *banked sequential,* which is described below).

Interleaved sequential is pipelined into three stages: *BTB, Cache,* and *Interchange-Valid.* There is bypass logic between the *BTB* and *Cache* stages so that the fetch pipeline latency for a mispredicted branch is two cycles, rather than three[1]. Since the typical length of instruction runs between branches is approximately four to six instructions, *interleaved sequential* does not perform well for high issue rates. This scheme can be enhanced by hardware that allows fetching to proceed across a branch.

3.2 Banked sequential

The *banked sequential* scheme is a modification of *interleaved sequential* to allow a limited amount of across-branch fetching. The hardware configuration is very similar to the former scheme (Figure 4). Alignment is possible only when the branch and its destination reside in different memory banks (inter-block branches). The hardware cannot handle intra-block branches. For a given fetch address, *banked sequential* finds the *likely successor address* then accesses the cache simultaneously for both the fetch block and its successor block. The likely successor address is determined by querying an interleaved BTB, as was done with the *interleaved sequential* scheme. Bank interference can occur if the successor block is in the same bank as the fetch block. In such a case, the successor block is not fetched.

Pipelining of banked sequential is similar to interleaved sequential, where the interchange switch and valid select form one stage of the three-stage pipeline. The BTB does not need to be queried again for the successor (prefetch) block. This is because as the fetch and successor blocks are being looked up in the cache (the second pipeline stage), the next instruction fetch queries the BTB with the successor

[1]The total misprediction penalty is the sum of the fetch misprediction penalty plus the number of cycles between when the branch is decoded and when it retires from the reorder buffer. This second component is instruction stream dependent and is modeled by the simulator.

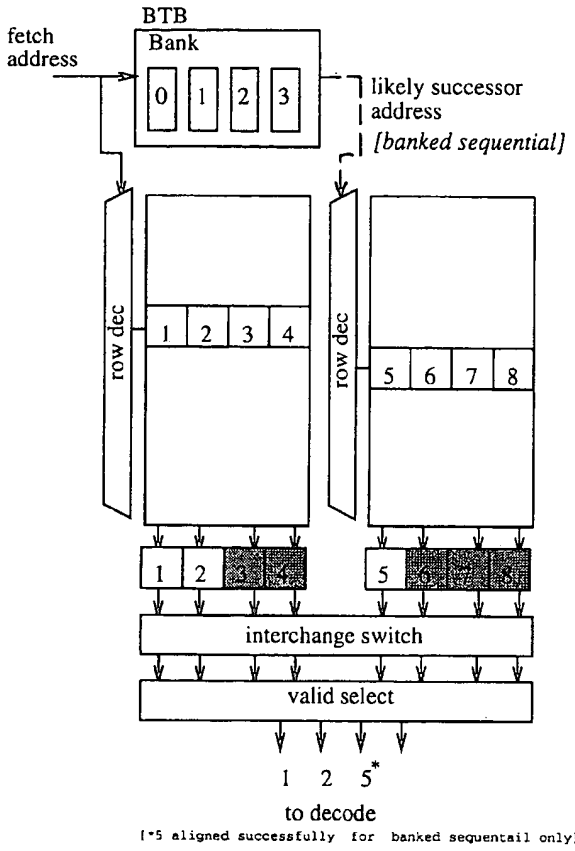


Figure 4: The *interleaved sequential* and *banked sequential* schemes.

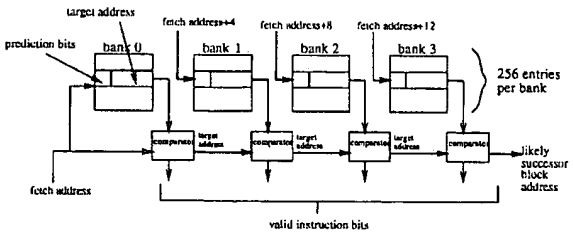


Figure 5: The interleaved BTB design (shown for PI4).

block address (the first stage). The BTB determines the successor block's valid bits with this overlapped cache/BTB access. Hence, the valid bits for the successor block are ready for use by *valid select* without two BTB queries.

Performance for *banked sequential* is limited by its inability to fetch across intra-block branches. The percentage of such branches to all taken branches for the workloads under study are shown in Table 2. For the PI4 machine (16-byte blocks), this percentage is small across all benchmarks except *compress* (14.58%). It increases dramatically as the block size increases. *Eqntott* increases from 6.13% to 29.26% from PI4 to PI8 (32-byte blocks). For PI12, almost half of the taken branches for *eqntott* (41.40%), *espresso* (45.68%) and *wave5* (41.73%) have their targets in the same block as the branch. This suggests the need for a mechanism to handle intra-block branches at high issue rates.

Table 2: Percentage of taken branches with target in the same block (*intra-block* branches).

Class	Benchmark	PI4	PI8	PI12
Int.	bison	6.05%	24.13%	30.81%
	compress	14.58%	14.59%	34.63%
	eqntott	6.13%	29.26%	41.40%
	espresso	1.40%	14.86%	45.68%
	flex	1.29%	3.88%	24.79%
	gcc	4.98%	14.08%	24.73%
	li	0.00%	5.74%	19.07%
	mpeg_play	0.70%	7.66%	11.96%
	sc	0.17%	11.02%	21.59%
FP	doduc	7.26%	11.85%	16.15%
	mdljdp2	0.26%	24.37%	66.10%
	nasa7	0.03%	0.06%	0.08%
	ora	0.01%	19.01%	23.16%
	tomcatv	0.08%	0.17%	13.97%
	wave5	2.71%	35.21%	41.73%

3.3 Collapsing buffer

The *collapsing buffer* scheme removes the useless instructions between an intra-block branch and it's target. It is an implementation designed to achieve *merging* [16], so that the target instruction follows the branch instruction in the decoder. This results in better decoder utilization and may also result in higher IPC.

The *collapsing buffer* scheme is shown in Figure 7. The BTB and cache are accessed in the same fashion as the previous two schemes. An additional buffer is

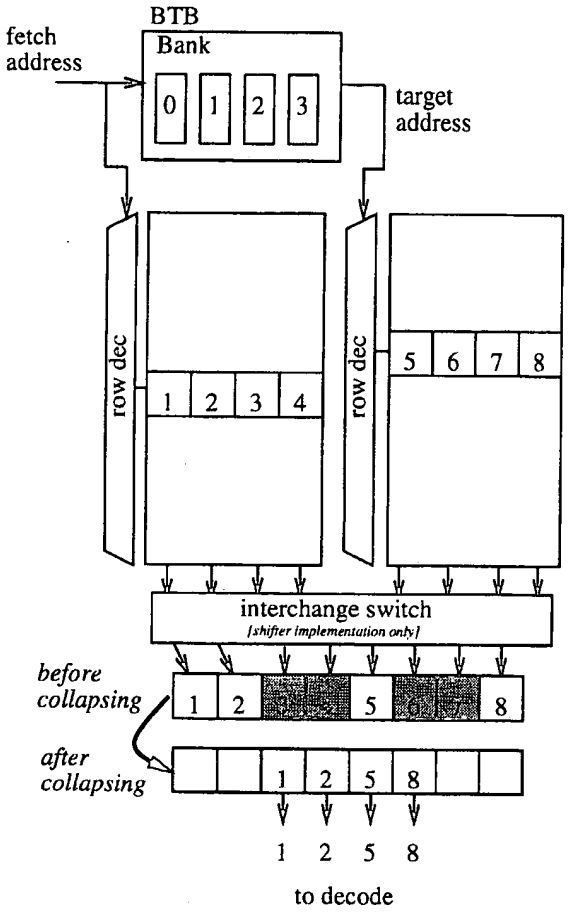


Figure 7: The *collapsing buffer* scheme.

added that collapses the gaps between valid instructions caused by intra-block branches. (Because of the capabilities of this buffer, the *valid select* logic of the previous two schemes has been removed.) Figure 8 details two possible implementations for the collapsing buffer. The first is a shifter-based implementation, and the second is a bus-based crossbar. The two implementations are open to tradeoffs based on area, speed and interconnect density. The crossbar implementation has the added advantage of being capable of handling backward branches, although this behavior was not supported by the controller modeled here.

Collapsing buffer is pipelined in a fashion similar to *banked sequential*. The crossbar implementation of the buffer removes the need for the *interchange switch* in addition to *valid select* logic. If traversal of the crossbar takes one cycle, the fetch misprediction penalty is two cycles. The shifter implementation will have a much higher misprediction penalty. Experiments with a penalty of three or more cycles produced little performance advantage for *collapsing buffer* over *banked sequential*, arguing against the shifter implementation (this is demonstrated below).

3.4 Performance of hardware schemes

The simulation results for *sequential, interleaved sequential, banked sequential,* and the *collapsing buffer* are shown in Figure 9(a) (integer benchmarks) and Figure 9(b) (floating-point benchmarks). Interleaving *sequential* provides a slight performance increase for both classes of benchmarks. Added fetch capabilities of the *banked sequential* and the *collapsing buffer* schemes provide distinct performance improvements, especially for integer benchmarks at higher issue rates. The floating-point benchmarks have well-behaved branches. Consequently, the performance of all the schemes for these benchmarks is relatively close for the PI4 machine. The need for more-sophisticated fetch mechanisms for floating-point code is more evident for the PI8 and PI12 machines, whose higher issue rates place a greater strain on the fetch unit.

The *collapsing buffer* is the most successful alignment mechanism across all processor designs. Floating-point benchmarks achieve almost perfect performance using this technique. Integer performance is also high, with IPCs very close to *perfect*. The justification for this scheme is provided by the difference in performance when compared to *banked sequential* for the PI12 machine. Here the gap in performance between the *collapsing buffer* and the other schemes is readily apparent.

Effective issue rate (EIR) is the rate at which instructions are successfully supplied to the decoders. For *perfect*, EIR is less than the ideal due to cache misses. For *sequential, interleaved sequential, banked sequential*, and the *collapsing buffer*, EIR is less than EIR(*perfect*) due to alignment failures. The ratio EIR/EIR(*perfect*) captures the ability of each of the schemes to align data. This metric is presented for each of the four schemes in Figure 10(a) (integer) and Figure 10(b) (floating-point). The *collapsing buffer* is the most-consistent scheme for delivering high EIR compared with EIR(*perfect*). It retains high performance in spite of increased issue rates from PI4 to PI12. The other schemes decrease in relative efficiency with approximately the same behavior from PI4 to PI12. (This is true for both integer and floating-point benchmarks.) This demonstrates that

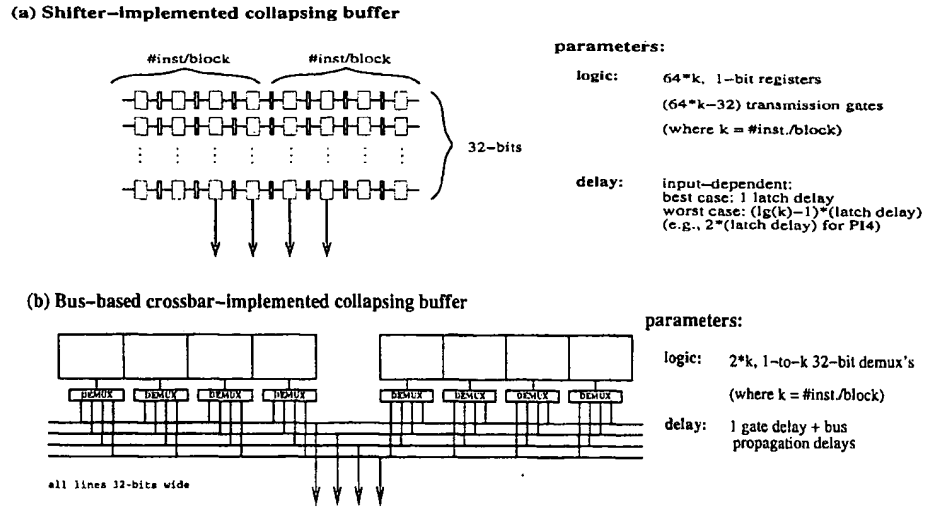


Figure 8: Design details of the collapsing buffer implemented (a) as a shifter, and (b) as a bus-based crossbar.

the *collapsing buffer* is a scalable alignment scheme, capable of delivering a high number of useful instructions in the presence of high issue rates.

In Section 3.3 it was mentioned that the shifter implementation of *collapsing buffer* does not provide much performance advantage over *banked sequential*. Figure 11 quantifies this observation. This figure is similar to Figure 9(a), except the *collapsing buffer* was simulated with a fetch misprediction penalty of three cycles. (This is perhaps the best-case performance for the shifter implementation.) *Banked sequential* actually performs slightly better than the *collapsing buffer/shifter* for PI4, and only slightly worse for PI12. This suggests that a low-misprediction-penalty implementation such as the crossbar is required to benefit from alignment using *collapsing buffer*[2].

4 The Effects of Compiler Optimizations

The hardware schemes presented above are limited by their ability to fetch across taken branches. Reduction of the number of non-sequential instruction accesses can lessen the impact of this limitation. The dynamic occurrence of taken branches can be reduced via compiler optimizations such as trace or superblock scheduling [17],[18]. These techniques reorder the code at compile time to form groupings of basic blocks that tend to execute sequentially. These larger groupings can be used to improve instruction cache performance, expand the scope of code scheduling, and enhance traditional optimizations [4],[7],[8],[19].

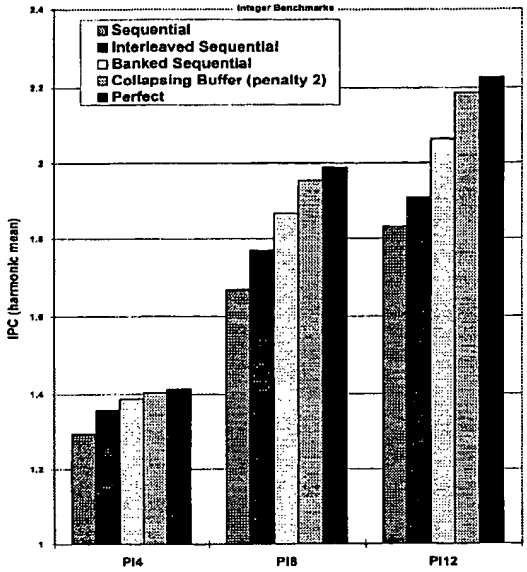
The effect of code reordering on the performance of the schemes was measured via simulation. Code reordering was performed on the benchmarks using trace selection and trace layout [7]. Six runs were performed for each integer benchmark. Each of the first five runs used a unique program input per run to generate profile statistics. These profiling inputs were taken from the input sets supplied by SPEC (or in the case of *li*, from student LISP assignments). An additional test input, not a member of the first five, was then used for the processor simulations. (SPECfp92 benchmarks were excluded since their code sequences are already highly-sequential in nature.)

The results of the simulations are presented in Figure 12. The figure also includes the performance of *sequential* and *perfect* without reordering (i.e., from the previous section), labeled as *sequential(unordered)* and *perfect(unordered)* in the figure. In general, code reordering significantly enhances performance. The success of code reordering can be attributed to a significant reduction in the number of taken branches. The percent reduction is shown in Table 3. The taken branches for a majority of the benchmarks are reduced by at least *20%*, and range from 15.72% for *li* to 44.2% for *compress*.

A detailed analysis of the data (Figure 12) reveals several interesting conclusions. *Sequential(reordered)*

[2]This observation is a function of the accuracy of the branch predictor.

(a)



(b)

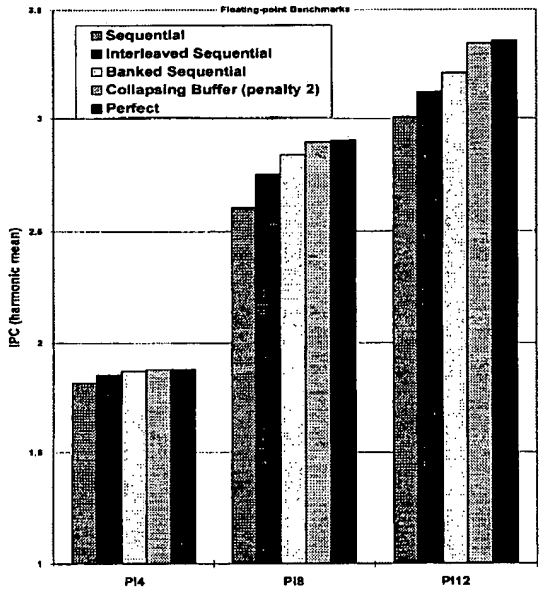
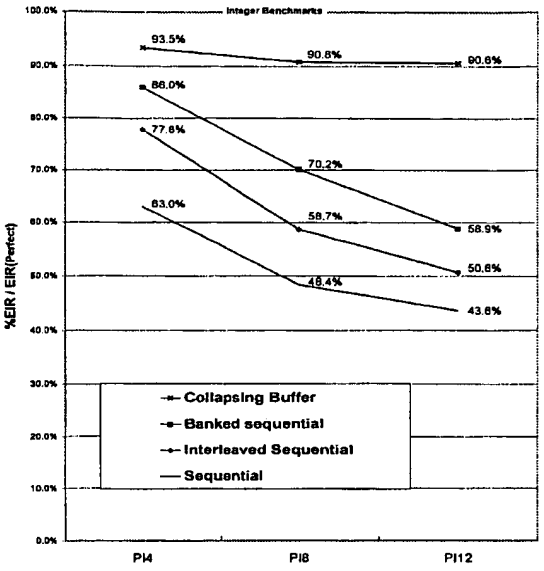


Figure 9: Performance of the alignment mechanisms for (a) integer, and (b) floating-point benchmarks.



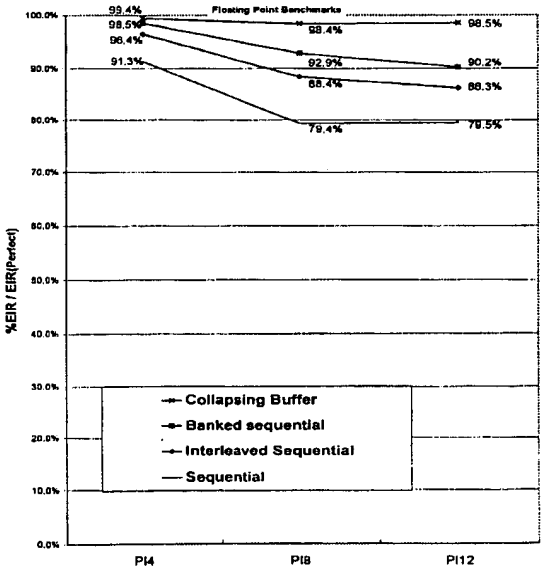


Figure 10: Percent EIR/EIR(*perfect*) of the alignment mechanisms for (a) integer, and (b) floating-point benchmarks.

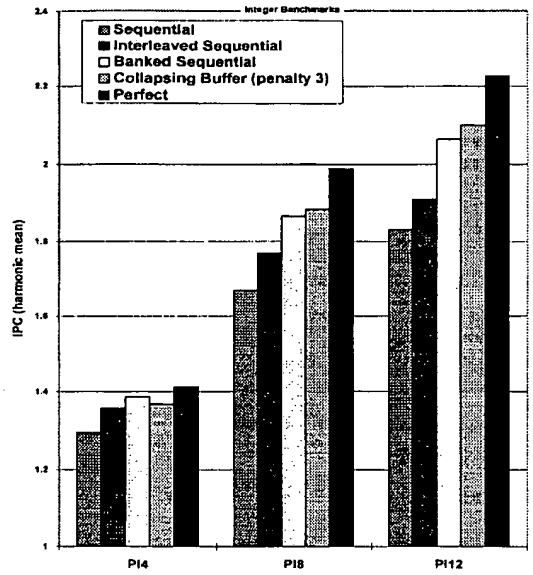


Figure 11: Performance of comparison of collapsing buffer assuming a three-cycle fetch misprediction penalty for integer benchmarks (all other schemes are shown with two-cycle penalties).

Table 3: Percent reduction in taken branches due to code reordering.

Benchmark	% Reduction
bison	25.26%
compress	44.20%
eqntott	24.52%
espresso	22.42%
flex	35.17%
gcc	37.20%
li	15.72%
mpeg_play	25.26%
sc	28.84%

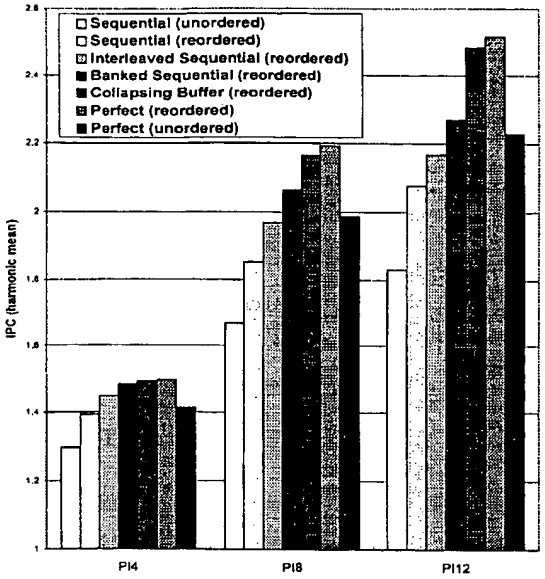


Figure 12: Performance of hardware schemes after code reordering.

achieves nearly the performance of *perfect(unordered)* for PI4. When reordered, the less-complicated *interleaved sequential* achieves comparable performance to *perfect(unordered)* across all three machine models. Hence, reordering can enhance the performance of *interleaved sequential* to match the performance of the hardware-only *collapsing buffer* scheme. However, when *collapsing buffer* is used with reordering, it nearly matches the performance of *perfect(reorderd)* from PI4 to PI12. This demonstrates that sophisticated compiler optimizations and sophisticated hardware combine to produce the highest performance for high issue rates.

4.1 Enhancing *sequential*

Reordering clearly enhances all hardware schemes. A compiler optimization to specifically enhance *sequential* is to align the traces by padding the end of each trace with *nops* to force the following trace to begin at a cache block boundary [8],[20]. This scheme is termed *pad-trace*. *Pad-trace* can increase the number of useful instructions in each fetched block. In addition, Fisher's trace selection algorithm places likely-taken branches at the end of traces. Since these branches transition to the beginning of other traces,

the inserted *nops* are seldom executed.

The disadvantage of both code reordering and *pad-trace* is that they require profile information, which is often hard to gather and requires additional steps when compiling code. (Hardware-based profiling techniques can remove many of these disadvantages, although their use was not studied in this paper. See [21].) An alternative to *pad-trace* is to pad all blocks without regard for trace membership. *Pad-trace* introduces significantly less *nops* than *pad-all*, as can be seen from Table 4.

Table 4: Degree of *nops* inserted for *pad-all* and *pad-trace* (expressed as percentage of *nops* vs. original code size).

block size 16B		
Benchmark	*pad-all*	*pad-trace*
bison	28.45%	2.22%
compress	29.53%	0.08%
eqntott	40.15%	7.17%
espresso	28.85%	5.60%
flex	27.75%	5.27%
gcc	32.31%	5.94%
li	33.20%	8.68%
mpeg_play	16.07%	3.45%
sc	37.89%	3.44%
block size 32B		
Benchmark	*pad-all*	*pad-trace*
bison	74.74%	5.35%
compress	74.98%	1.85%
eqntott	98.95%	16.77%
espresso	74.05%	12.93%
flex	67.65%	13.47%
gcc	80.33%	14.23%
li	80.33%	19.20%
mpeg_play	43.11%	8.87%
sc	90.71%	8.29%
block size 64B		
Benchmark	*pad-all*	*pad-trace*
bison	183.6%	12.28%
compress	190.8%	4.06%
eqntott	254.9%	41.37%
espresso	196.2%	30.50%
flex	173.6%	33.01%
gcc	214.0%	34.49%
li	225.1%	41.85%
mpeg_play	105.0%	21.18%
sc	237.8%	20.18%

The performance of *sequential* when augmented using *pad-all* and *pad-trace* is shown in Figure 13. Of the two, *pad-trace* achieves marginally higher performance improvement over its counterpart, *sequential(reordered)*, than *pad-all* achieves over *sequential(unordered)* for PI4. *Pad-all* achieves gains only

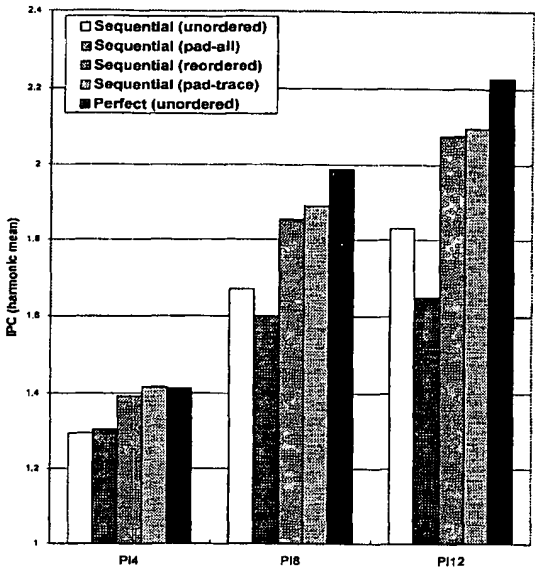


Figure 13: Performance of *pad-all* and *pad-trace* for *sequential*.

for PI4, and experiences poor performance for processors using a larger cache size. This is due to the reduction in cache locality caused by excessive *nop* insertion. In general, *pad-all* appears to be unjustified even for PI4, since its benefit is more than offset by code expansion (Table 4). The code expansion for *pad-trace* is minor, justifying it as a refinement of code reordering.

5 Concluding Remarks

The results presented in this paper demonstrate the need for efficient instruction alignment in order to support highly-parallel microarchitectures, such as PI8 and PI12. It appears that some fetch mechanisms such as *interleaved sequential* or *banked sequential* are also required for PI4 (which is similar in structure to several next-generation processors). The most robust scheme across all architectures studied was the *collapsing buffer*. The evidence for this is presented in the EIR/EIR(*perfect*) data of Figure 10. The *collapsing buffer* consistently aligns instructions at least 90% of the time.

The frequency of short branches within the same block motivated the design of the *collapsing buffer*.

Compiler-based techniques such as trace layout (reordering) can reduce this phenomenon by eliminating many taken branches. The data shows that these techniques can significantly enhance all schemes. For example, code reordering can enhance the performance of *interleaved sequential* to nearly match that of a hardware-only *collapsing buffer* approach for PI12. This also suggests that these techniques are applicable to existing machines. Padding with *nops*, either used with reordering or used separately, produced only marginal improvements for *sequential* (the remainder of the hardware schemes were not significantly enhanced by padding). The best overall solution is to combine the highest-performance hardware scheme (*collapsing buffer*) with code reordering.

It remains to be seen what effect branch prediction accuracy has on the misprediction penalty when designing a pipelined *collapsing buffer*. Other, more sophisticated predictors do exist that have been designed for machines with high misprediction penalty [9]. Depending on the complexity of this branch prediction hardware, a shifter-based implementation of *collapsing buffer* may be viable.

Acknowledgements

We would like to thank Sumedh Sathaye, Ashutosh Singla, Prashant Maniar and the anonymous reviewers for their comments and suggestions on improving this paper. This research has been supported by AT&T.

References

[1] L. Gwennap, "MIPS R10000 uses decoupled architecture," *Microprocessor Report*, Oct. 1994.

[2] A. Agarwal, "UltraSPARC: A new era in SPARC performance," in *1994 Microprocessor Forum Proceedings*, Oct. 1994.

[3] M. Slater, "AMD's K5 designed to outrun Pentium," *Microprocessor Report*, Oct. 1994.

[4] S. McFarling, "Program optimization for instruction caches," in *Proc. Third Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 183–191, Apr. 1989.

[5] D. B. Whalley, "Fast instruction cache performance evaluation using compile-time analysis," in *Proc. ACM SIGMETRICS '92 Conf. on Measurement and Modeling of Comput. Sys.*, (Newport, RI), pp. 13–22, June 1992.

[6] C. L. Mitchell and M. J. Flynn, "The effects of processor architecture on instruction memory traffic," *ACM Trans. Comput. Sys.*, vol. 8, pp. 230–250, Aug. 1990.

[7] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 242–251, May 1989.

[8] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[9] T. Yeh, *Two-level adaptive branch prediction and instruction fetch mechanisms for high performance superscalar processors.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 1993.

[10] B. Calder and D. Grunwald, "Fast & accurate instruction fetch and branch prediction," in *Proc. 21st Ann. International Symposium on Computer Architecture*, pp. 2–11, Apr. 1994.

[11] S. McFarling, "Combining branch predictors," WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[12] L. Gwennap, "PA-8000 combines complexity and speed," *Microprocessor Report*, Nov. 1994.

[13] S. Weiss and J. E. Smith, *POWER and PowerPC.* San Francisco, CA: Morgan Kaufmann, 1994.

[14] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[15] S. P. Song and M. Denman, "The PowerPC 604 RISC microprocessor," tech. rep., Somerset Design Center, Austin, TX, Apr. 1994.

[16] M. Johnson, *Superscalar microprocessor design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[17] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[18] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[19] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software–Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[20] M. Smotherman, S. Chawla, S. Cox, and B. Malloy, "Instruction scheduling for the Motorola 88110," in *Proc. 26th Ann. Int'l. Symp. on Microarchitecture*, (Austin, TX), pp. 257–262, Dec. 1993.

[21] T. M. Conte, B. A. Patel, and J. S. Cox, "Using branch handling hardware to support profile-driven optimization," in *Proc. 27th Ann. International Symposium on Microarchitecture*, (San Jose, CA), Nov. 1994.

Block-level prediction for wide-issue superscalar processors

Dutta, S. Franklin, M.
Dept. of Electr. & Comput. Eng., Clemson Univ., SC;

This paper appears in: **Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP. IEEE First International Conference on**
Meeting Date: 04/19/1995 -04/21/1995
Publication Date: 19-21 April 1995
Location: Brisbane, Qld. , Australia
On page(s): 143-152 vol.1
Volume: 1, References Cited: 14
INSPEC Accession Number: 5073688

Abstract:
Changes in control flow, caused primarily by conditional branches, are a prime impediment to the performance of wide-issue superscalar processors. This paper investigates a block-level prediction scheme to mitigate the effects of control flow changes caused by conditional branches. Instead of predicting the outcome of each conditional branch individually, this scheme predicts the target of a sequential block of instructions, thereby allowing the superscalar processor to go past multiple branches per cycle. This approach is evaluated using the MIPS architecture, for 8-way and 12-way superscalar processors, and an improvement in effective fetch size of approximately 15% and 25%, respectively, over identical processors that use branch prediction is observed. No appreciable difference in the prediction accuracy was observed, although block-level prediction predicted one out of four outcomes

Index Terms:
MIPS architecture block-level prediction conditional branches instruction sets microprocessor chips pipeline processing wide-issue superscalar processors MIPS architecture block-level prediction conditional branches instruction sets microprocessor chips pipeline processing wide-issue superscalar processors

Block-Level Prediction for Wide-Issue Superscalar Processors

Simonjit Dutta and Manoj Franklin
Department of Electrical and Computer Engineering
Clemson University
102 Riggs Hall
Clemson, SC 29634-0915, USA
Email: simonjd@eng.clemson.edu, mfrankl@eng.clemson.edu

Abstract

Changes in control flow, caused primarily by conditional branches, are a prime impediment to the performance of wide-issue superscalar processors. This paper investigates a block-level prediction scheme to mitigate the effects of control flow changes caused by conditional branches. Instead of predicting the outcome of each conditional branch individually, this scheme predicts the target of a sequential block of instructions, thereby allowing the superscalar processor to go past multiple branches per cycle. This approach is evaluated using the MIPS architecture, for 8-way and 12-way superscalar processors, and an improvement in effective fetch size of approximately 15% and 25%, respectively, over identical processors that use branch prediction is observed. No appreciable difference in the prediction accuracy was observed, although block-level prediction predicted one out of four outcomes.

1 Introduction

Superscalar processors are becoming very popular now. Almost all of the recent microprocessor releases have applied the superscalar technique in some way or other [1] [12]. As more and more transistors can be placed in a chip, the issue width of these processors also keeps increasing steadily. To sustain a high issue rate, a superscalar processor fetches and decodes multiple instructions per cycle from a single flow of control. Needless to say, the sustained instruction issue rate of a superscalar processor will never exceed the average number of (useful) instructions it fetches per cycle.

Fetching multiple instructions in a cycle is not difficult so long as the fetched instructions are from a straight line piece of code. Conditional branches are the primary impediment to fetching a large number of instructions per cycle, because they are frequent (roughly one in every 5 instructions, on the average), and can potentially alter the flow of control. Researchers realized this problem fairly early on, and the solution proposed was to predict the outcome of the branch before it was resolved, and to fetch and execute instructions in a speculative manner from the predicted target. If the prediction is later found to be incorrect, the effects of the instructions executed after the branch are nullified, and execution continues along the correct path. Branch prediction techniques have now improved to the point of getting about 90%-96% prediction accuracies for non-numeric programs [9] [14].

One problem that cannot be solved by high-accuracy predictions alone is that a branch's identity and target are not known until the branch instruction is decoded. If the prediction is fall-through, then this is not a problem; if the branch is predicted to be taken, then this causes a 1-cycle bubble in the fetch pipeline. Furthermore, if at most a single branch prediction is made per cycle, the instruction fetch mechanism can fetch no more than 5 instructions per fetch attempt on the average. Both these factors severely limit the the instruction level parallelism that can be exploited by superscalar processors. It is clear that to overcome the above bottleneck, either (i) the number of branches encountered by the hardware has to be reduced (through techniques such as *guarding* [6] [11]) or (ii) the target information has to be associated with higher-level blocks (as opposed to associating them with the branches), and the outcomes of multiple branches need to be predicted in a cycle.

0-7803-2018-2/95/$4.00 1995 IEEE

This paper focuses on the second approach.

The approach considered in [10], called *control flow prediction*, is to predict the target of a subgraph of the control flow graph of the executed program. By predicting one of the targets, the control flow predictor specifies the next subgraph to be (speculatively) executed, and thereby goes past one subgraph a cycle. In the type of control flow prediction considered in [10], a subgraph can have an arbitrary number of branches encompassed within, and can therefore have arbitrary structure, length, and number of targets. Therefore, the instructions in a subgraph are most likely fetched sequentially. Such a control flow prediction scheme, with unrestricted subgraph structures, is more apt for execution models that pursue multiple flows of control, such as the *multiscalar processor* [2], and is not the subject of this paper.

For processors that follow the superscalar model of execution, we need to restrict the subgraph structure, and identify a path within the subgraph to fetch instructions from. The approach considered in [13] is to use tree-like subgraphs, and select a path through the tree-like subgraph by performing 2-3 branch predictions in a cycle, using a high accuracy, 2-level branch predictor [14]. For a given subgraph address, the prediction mechanism performs these predictions before even determining the addresses of the intermediate conditional branches encompassed within the subgraph. Based on the prediction values, a *Branch Address Cache* provides the intermediate fetch addresses. The instruction fetch mechanism then fetches instructions from these multiple targets. Because the intermediate branch addresses are not known when the predictions are made, this scheme uses the condensed history of all branches to make its decisions.

This paper considers a sequential block of instructions (with multiple branches encompassed within) as a subgraph. Furthermore, instead of predicting the outcome of each conditional branch in the block, a single prediction is made about which of the possible targets will be taken, and instruction fetch is continued from that target in the next cycle. The advantage of this scheme is that the predictor and the instruction fetch mechanism need not be any more complex than those in a superscalar processor implementing branch prediction.

This paper is organized as 5 sections. The introduction has highlighted the need to go beyond predictions at the branch level and to predict the outcome of multiple branches per cycle. Section 2 details the operation of block-level prediction, and discusses techniques for carrying out this scheme. Section 3 calculates the hardware cost of the block-level prediction scheme. Section 4 presents an experimental evaluation of block-level prediction using the MIPS architecture. Section 5 presents a summary and the conclusions.

2 Block-Level Prediction

2.1 General Operation

The central idea of the new predictor is to make control flow predictions at the block level, where a block is a sequential block of instructions. That is, instead of predicting the outcome of each branch (to decide the speculative execution path), predict the target of a sequential block of instructions.

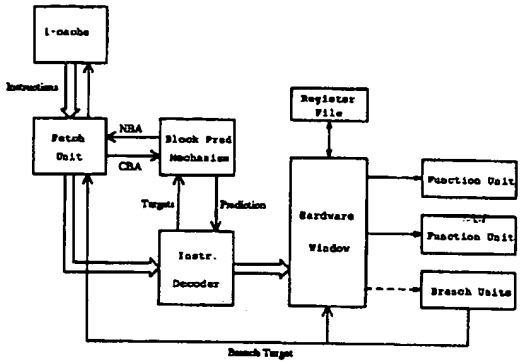


Figure 1: Block Diagram of a Processor with Block-Level Prediction

Figure 1 gives a block diagram showing the relationship between the instruction cache, the block predictor, the instruction fetch unit, the decode unit, the hardware window, and the functional units in a superscalar processor. The fetch unit has the address of the next block to be fetched, and sends this address to the block predictor. The block predictor makes a prediction to select an outcome and the corresponding target, and sends the outcome and the target to the fetch unit. The fetch unit fetches this block from the multi-ported i-cache, and forwards the fetched instructions to the decoder. If the fetch unit encounters a particular block for the first time, it also tells the decoder to send that block's targets to the prediction mechanism for storage.

The decoder decodes the instructions fetched in the previous cycle by the fetch unit, taking into con-

sideration the prediction made earlier (by the predictor) on the outcome of that block. It discards any instructions that succeed a conditional branch that is indirectly predicted to be taken. The decoded instructions are sent to the hardware window structure, which serves as a platform for performing dynamic scheduling among the decoded instructions. Multiple functional units are provided to execute multiple instructions per cycle. Among these, the branch units carry out the execution of control-changing instructions. When a control-changing instruction is executed, a check is made to see if the actual outcome is in line with the earlier block prediction; if there is a discrepancy, the hardware window discards the instructions following the point of discrepancy. It also sends the misprediction information to the fetch unit, which then starts fetching from the correct target.

2.2 How Many Targets Per Block?

Before proceeding to study how exactly the block predictions are made, we shall pause briefly to examine some restrictions that we need to impose on the size of our blocks. We can consider 2 attributes when discussing the size of a sequential block: (i) the number of control flow outcomes a block can have, and (ii) the number of instructions in a block.

The first attribute is decided by the number of branches encompassed in a block. If a block has only 2 outcomes, then only one branch is contained in a block, and only one branch is effectively predicted per cycle. If a single branch is effectively predicted per cycle, the average number of useful instructions fetched per cycle can never exceed the average number of instructions between consecutive branches, which, for non-numeric codes, is only about 5. On the other hand, if a block has a large number of possible outcomes, then the prediction mechanism has to choose one outcome from many, resulting in potentially lower prediction accuracies. Each time a misprediction is detected, the subsequent instructions (the ones after the mispredicted branch) in the hardware window have to be discarded. Thus, although a larger block is fetched at a time, the poor prediction accuracy could potentially reduce the performance compared to the case when a block has fewer outcomes. As a trade off, for our studies in this paper, we allow up to 4 outcomes per block. This will allow up to 3 conditional branches to be enclosed within a block[1]. The outcomes of a block are encoded as follows: The conditional branches in a block are assigned numbers {1, 2, 3}, depending on their sequential order of appearance in the block. The outcome obtained when conditional branch i is taken is encoded as *outcome* i. The outcome obtained if all conditional branches follow the fall-through path is encoded as *outcome* 0. Associated with each possible outcome, there is a single target. Multiple outcomes can have the same target, however. Figure 2 illustrates the encoding scheme used for the outcomes of a block. In the figure, CBi represents the i^{th} conditional branch in a block, and the numbers {0, 1, 2, 3} denote the possible outcomes of a block.

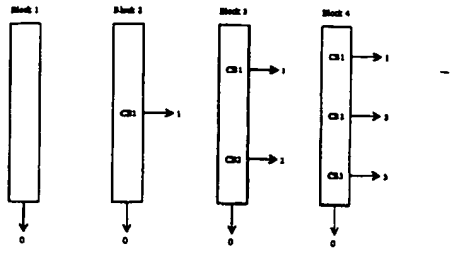


Figure 2: Encoding of Block Outcomes for Blocks with 1, 2, 3, and 4 Outcomes

Now consider the second attribute, namely the number of instructions in a block. Ideally, the fetch mechanism should be able to fetch all instructions from a block in one cycle. This will allow one block prediction to be carried out every cycle. Thus, the length of a block cannot exceed f, where f is the maximum fetch size of the superscalar fetch mechanism.

2.3 Prediction Mechanism

Let us now return to our discussion of the means for carrying out block predictions. The block predictions are made with the help of a *2-level block predictor*[2], similar in spirit to the 2-level branch predictor described in [14].

Figure 3 shows the block diagram of the various parts within our 2-level block predictor. It consists of two main structures—the *Block History Table (BHT)* and the *Pattern History Table (PHT)*. The BHT is

[1] Restricting the number of conditional branches in a block to 3 seems to be a good choice, as indicated by the results given later in Table 3.

[2] It is not mandatory to use a 2-level prediction scheme for implementing block prediction. We describe an implementation using the 2-level approach, as it tends to give high prediction accuracies.

a cache-like structure that stores information relevant to the blocks that have been encountered in the recent past. Each BHT entry has 3 fields—Tag, Targets, and Block History Pattern. The Tag field is for uniquely identifying the block currently mapped to that entry. The Targets field stores the targets of the block, and the Block History Pattern field stores the last p outcomes ($p = 6$ in the figure) of the block as a $2p$-bit pattern. Because there are 4 possible outcomes for a block, 2 bits are required to store each outcome. The secondary level of the predictor is the PHT. For each possible $2p$-bit pattern, a condensed history of the previous outcomes corresponding to the pattern is recorded in the PHT by means of 4 independent up/down counter values $\{C_0, C_1, C_2, C_3\}$.

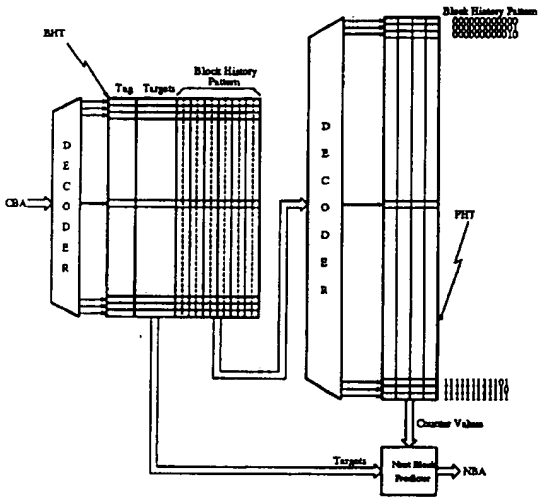


Figure 3: Block Diagram of 2-Level Block Predictor

When the instruction fetch unit makes a prediction request for a particular block address, the appropriate entry in the BHT is selected. The tag value of the entry is checked to determine if the entry corresponds to the current block address. If so, the entry's block history pattern is used to select one of the entries from the PHT. The PHT entry contains 4 count values, and the prediction mechanism determines the maximum count value. The outcome corresponding to the maximum count value is selected as the next prediction value. (If there is a tie between two or more count values for the maximum position, the selector could choose the last outcome as the next prediction, if the last outcome is also stored in each PHT entry.) The BHT entry also contains the targets corresponding to the current block. Depending on the prediction value selected, a target is chosen from the targets available in the selected BHT entry, and sent to the instruction fetch unit as the next block address.

When a prediction is made, in addition to sending the next target of a block, the predictor also updates the history of that block. The PHT entry's counter values are updated in the following manner. The count value corresponding to the predicted outcome is incremented by 3 (or less if incrementing leads to the counter value reaching its maximum allowed value), and all other count values are decremented by 1 (if their count values are not already 0). Finally, the block history pattern corresponding to that block address is shifted left by 2 bit positions, and the latest outcome is entered in the 2 rightmost bit positions (*i.e.*, the bit positions left vacant by the left shift). Notice that these updates to the BHT and PHT entries, done immediately after making a prediction, are done in a speculative manner. By doing so, up-to-date history is available for making future predictions. (As the prediction accuracy of 2-level prediction schemes is very high, using the speculatively updated history would result in better prediction accuracy than using obsolete history to make the predictions.) If a prediction is later found to be incorrect, the speculative updates done to the BHT and PHT entries corresponding to the blocks discarded by the recovery mechanism are setback.

2.3.1 Handling New Blocks

When a block B is encountered by the fetch unit for the first time, or if its entry in the BHT was replaced by another block, the BHT will not have an entry for B. If the fetch unit requests a prediction for B in such a situation, the predictor selects a prediction value of 0, and returns a target value equal to B's address + f, where f is the maximum fetch size, to the fetch unit. When this block is fetched, the fetch unit requests the decoder to send the correct targets to the BHT for storage.

2.4 Determination of New Target at Times of Recovery

When a prediction is made for a block, the outcomes of the conditional branches contained in the block also get indirectly predicted. When the predicted outcome of a conditional branch is later found to be incorrect, recovery actions are initiated. These actions include clearing from the instruction window all instruc-

tions that have been fetched after the mispredicted branch. The recovery process also includes finding a new block prediction for the block that contains the mispredicted branch. If the branch was predicted to be fall-through and ends up being taken, then finding the new block prediction is straightforward; the new block prediction is given by the *branch number b* of the mispredicted branch within its block. This case is illustrated in Figure 4, for $b = 2$. The first figure shows a block whose outcome was originally predicted to be 3, as shown by the thick line. The corresponding predictions for the conditional branches CB1, CB2, and CB3 are fall-through, fall-through, and taken, respectively. Assume that CB2 when executed ended up being taken. The new prediction for the block is given by 2, as shown by the thick line in Figure 4(ii).

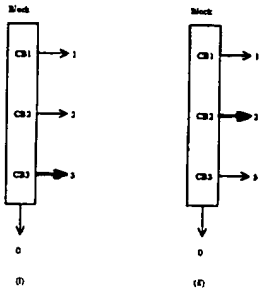


Figure 4: Recovery when Branch CB2 was Predicted to be Fall-through

Now consider the case when a branch was predicted to be taken, but ends up being fall-through. Finding a new prediction in this case is not so straightforward, as there could be more than one possible outcome for the block when the fall-through path of that branch is taken (if there are one or more conditional branches in the fall-through path). In such a scenario, the recovery mechanism requests the prediction mechanism to give a prediction greater than b or equal to 0 (corresponding to all conditional branches within the block taking the fall-through path). This is illustrated in Figure 5, for $b = 2$. The first figure shows a block whose outcome was originally predicted to be 2, as shown by the thick line. The corresponding predictions for the conditional branches CB1 and CB2 are fall-through and taken, respectively. Assume that CB2 when executed ended up being fall-through. The new prediction for the block could be either 3 (as shown by the thick line in Figure 5(ii)), or 0 (as shown by the thick line in Figure 5(iii)).

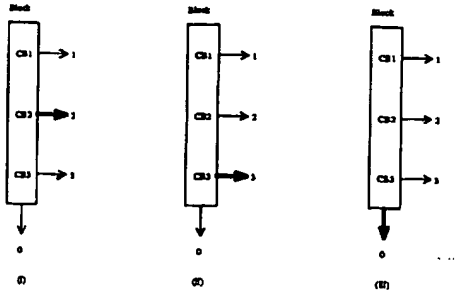


Figure 5: Recovery when Branch CB2 was Predicted to be Taken

2.5 Advantages of Block Prediction

Block-level prediction, as described here, has several advantages. First, it helps to predict the outcome of multiple branches per cycle. This is essential to improve the performance of wide-issue superscalar processors for non-numerical programs. Second, the fetch mechanism has to fetch only a sequential block of instructions. This is considerably easier than simultaneously fetching variable number of instructions that start from multiple addresses. Third, only a single prediction is made per cycle. Thus, the prediction mechanism need not be any more complex than an ordinary branch prediction scheme. One potential limitation of this technique (as will become evident in Section 4, where we present the experimental results) is that the *effective fetch size* (i.e., the number of (useful) instructions entered into the hardware window from a block) can be somewhat smaller than the maximum fetch size because of branches that are predicted to be taken. However, the compiler can attempt to modify the branch opcodes or schedule the instructions in such a manner that the conditional branches in the beginning of frequently-executed blocks most likely take the fall-through path (as done in [3] for the IBM RS 6000). This will cause most of the instructions of those blocks to be executed, thereby resulting in large effective blocks. Techniques for doing this is beyond the scope of this paper.

3 Hardware Cost

Before proceeding to do an experimental evaluation of the performance of block-level prediction, let us calculate the hardware cost for implementing the 2-level block predictor. The two hardware-intensive components of the block predictor are the BHT and the

PHT. Let us calculate the hardware costs for each of these components.

Let the number of entries in the BHT be N. The Tag field of a BHT entry requires as many bits as to uniquely identify a block address, and depends on N, the block address width (W), and the type of mapping used to select a BHT entry. In any case, the number of bits required for the Tag field will not exceed W. The Targets field of a BHT entry needs to store 4 target addresses (each requiring W bits). Lastly, the Block History Pattern field of a BHT entry stores the past p outcomes of the block mapped currently to that BHT entry, and requires $2p$ bits. Thus, the maximum number of bits required to implement the BHT is $N \times (5W + 2p)$ (which works out to be 10.75Kbytes for $N = 512$, $W = 32$, and $p = 6$).

Next, let us look at the number of bits required for the PHT. Because each Block History Pattern has $2p$ bits, the PHT has 2^{2p} entries. Each PHT entry needs to store 4 count values, each requiring, say c bits. The total number of bits for implementing the PHT is therefore given by $4c \times 2^{2p}$ (which works out to be 8Kbytes for $c = 4$ and $p = 6$). Alternate methods that reduce the hardware cost is a subject for future research.

4 Experimental Evaluation

The previous sections described block-level prediction, its implementation, and hardware cost. This section presents the results of an empirical study of its performance. For comparison purposes, we also evaluate the 2-level branch prediction scheme [14], augmented with a BTB to avoid the 1-cycle penalty due to branches that are predicted to be taken.

4.1 Experimental Setup

The performance studies are conducted using the MIPS instruction set architecture (ISA) [4], a representative of the class of streamlined ISAs that have emerged recently [5]. There are several implementations of the MIPS architecture, namely, R2000, R3000, R4x00, R6000, R8000, and R10000. (Of these, R8000 and R10000 are superscalar processors.) An important aspect of the MIPS architecture is the one cycle delay slot for all control-changing instructions such as branches, jumps, and calls; i.e., the instruction following a control-changing instruction is always executed if the control-changing instruction is executed. This architected delay slot presents a problem while deciding the block boundaries. A delay slot instruction must be filled in the same block that contains its corresponding control-changing instruction. If the last instruction in a fetch block A is a control-changing instruction C, then there is no vacant slot to accommodate its delay slot instruction in the same block. In such a situation, it is difficult to consider C's target to be one of the targets of block A, because control does not go to that target after the execution of block A. On the other hand, this target cannot be attached to the block containing the delay slot, say block B, because block B could potentially be entered through a different path. To avoid this problem, if the last instruction in a fetch block is a control-changing instruction, then it is not included in that block; instead that instruction is grouped together with its delay slot instruction in the subsequent block.

4.1.1 Simulation Tool and Benchmarks

All data reported in this paper are gathered with a simulator that accepts programs compiled for a MIPS-based DECstation and simulates their execution, keeping track of relevant information on a cycle-by-cycle basis. *The simulator models speculative execution, and is not trace-driven.* Because the entire processor is modeled in the simulator, the simulation results reflect reality to a very close extent. System calls made by the simulated programs are handled with the help of traps to the operating system. The collected results therefore exclude the code executed during system calls, but include all other code portions, including the library routines.

For benchmarks, we use 5 programs from the SPEC '92 integer suite: *compress* with input file *in*, *eqntott*, *espresso* with input file *bca*, *gcc* with input file *stmt.i*, and *xlisp* with input file *li-input.lsp*, which are all integer-intensive programs written in C. All programs were compiled using the MIPS C compiler; the SPEC benchmarks were compiled using the optimization flags distributed in the SPEC benchmark makefiles. The benchmarks were simulated to completion or up to 400 million instructions, depending on whichever occurred first.

4.1.2 Performance Metrics

For measuring performance, execution time is the sole metric that can accurately measure the performance of an integrated software-hardware computer system. However, this metric requires many implementation assumptions, including the exact hardware configu-

ration, functional unit latencies, etc. Other metrics, such as instruction issue rate and number of instructions in hardware window, also require implementation assumptions that limit the utility of results. Further, these metrics may not give much insight about the performance of the fetch mechanism. Therefore, for this study, we use the following (indirect) metrics, which provide good information about the fetch mechanism's performance: (i) *effective fetch size (EFS)*, (ii) *block prediction accuracy (BlPA)*, (iii) *branch prediction accuracy (BPA)*, and (iv) *block termination cause (BTC)*.

The first metric indicates the number of *useful* instructions fetched in each fetch attempt by the fetch unit. That is, instructions that are fetched but discarded because of a conditional branch being predicted to be taken, are not counted in this metric. Furthermore, useless instructions that are discarded due to incorrect speculative execution are also not counted while calculating this metric. Notice also that, if a block had to be fetched in 2 (or 3) attempts because of mispredictions, then it is counted as 2 (or 3) fetch attempts. Thus, EFS takes into account the effect of the inaccuracies in block prediction. This is an important metric because increasing its value is the primary goal of block-level prediction. Fetching more instructions per cycle gives the superscalar hardware window more opportunities to look for and exploit instruction-level parallelism.

The second metric of interest to us is the BlPA. It indicates the fraction of times a block prediction gives the correct target. Although the first metric completely captures the effect of block prediction inaccuracies, this metric throws more light on why the performance of the predictor is better or worse. We are also interested in knowing if making a 4-way block prediction instead of a 2-way branch prediction results in a poor prediction accuracy. The third metric, namely the BPA, indicates the fraction of times a branch (indirectly) gets correctly predicted when block prediction is performed. This metric is highly dependent on the second metric, but is helpful in carrying out comparisons with the BPAs obtained with other prediction techniques.

The last metric, namely the BTC, is highly instructive in that it helps researchers to chart directions for future research.

4.1.3 Caveat

It is important to make note of a caveat in our evaluation. The performance of any computer system is highly dependent on the compiler and the type of static scheduling done. All our studies are carried out with code compiled for a single-issue processor. The major implication of this decision is: no block-level prediction-specific optimizations were performed on the code; the code is scheduled for a single-issue processor, which can have serious implications on the performance of block-level prediction. In that sense, the performance results presented here should only be viewed as a realistic starting point. Further doctoring of the code, especially to make branches take the fall-through path more often than the taken path as in [3], will certainly improve the performance of block-level prediction further.

4.1.4 Default Parameters for the Study

We have attempted to do a reasonable study by varying the important parameters independently. When a parameter is varied, the rest of the parameters are kept fixed at their default values, given below:

- *Maximum Fetch Size (f)* The default value is 12 instructions.
- *Branch Predictor:* This scheme is simulated with a *PAg* 2-level branch predictor, with a pattern size (p) of 6. The BHT has 1024 entries, and is direct mapped. The PHT entries consist of 3-bit up/down saturating counters. The predictions are done based on the start address of each fetch block, so that the predictions can be done before decoding the instructions. *Therefore, strictly speaking, this scheme is a block-level predictor, with each block restricted to 2 outcomes.*
- *Block Predictor:* This scheme is simulated with a *PAg* 2-level block predictor. The BHT is also direct mapped, and the default value for the number of BHT entries is 1024. Each PHT entry has four 4-bit up/down saturating counters. The default pattern size is 6.

4.2 Experimental Results

4.2.1 Effective Fetch Size

Table 1 shows the EFS obtained with the default branch prediction scheme and the block prediction scheme, for $f = 8$ and $f = 12$. The first 2 columns of numbers present the EFS obtained with branch prediction for $f = 8$ and $f = 12$, respectively, and the

subsequent 2 columns present the same for block prediction. The last 2 columns present the percentage improvement in EFS when using block-level prediction, compared to the branch prediction case, for $f = 8$ and $f = 12$, respectively.

Program	Br. Pred.		Block Pred.		% Incr.	
	f=8	f=12	f=8	f=12	f=8	f=12
compress	4.72	5.74	5.84	7.22	23.73	25.78
eqntott	3.54	3.54	4.60	4.98	29.94	40.68
espresso	4.54	5.08	4.96	5.84	9.25	14.96
gcc (cc1)	4.55	5.12	4.75	5.62	4.40	9.77
xlisp	4.59	4.88	5.01	5.92	9.15	21.31

Table 1: Effective Fetch Size for $f = 8$ and $f = 12$

Let us look at the results of this table in some detail. The percentage improvement in EFS for $f = 8$ varies from 4.4% (for gcc) to 29.9% (for eqntott). When f is increased to 12 instructions, the percentage improvement for block-level prediction over branch prediction has improved, and varies from 9.8% (for gcc) to 40.7% (for eqntott). For all benchmarks (except compress), the percentage improvement in EFS has improved significantly, when f was increased from 8 to 12 instructions. This is because the branch predictor is fundamentally limited by the basic block size, and therefore does not improve much when f is increased. The block-level predictor, on the other hand, tends to gain when f is increased[3].

4.2.2 Prediction Accuracy

Next, let us look at the prediction accuracies obtained. Table 2 shows the prediction accuracies that we obtained for $f = 12$. The first column of numbers gives the BPA obtained with the branch prediction scheme. The second column of numbers gives the BlPA obtained with the block-level prediction scheme. The last column of numbers gives the BPA that was obtained when block-level prediction was carried out. Let us look at these results closely. The first thing to notice is that the two-level prediction scheme has worked very well for both branch prediction and block prediction schemes. Second, the BPA obtained for the block-level prediction scheme is very close to the BPA obtained for the branch prediction case. This result is very encouraging, because it shows that predicting one out of 4 outcomes has provided more or less the same accuracy as predicting one out of 2 outcomes.

[3]This increase is likely to taper off at some point, due to conditional branches that cause control to flow out of blocks from the middle (c.f. Table 3).

(In the block prediction scheme results for compress, the BlPA is higher than the BPA because BlPA includes blocks that do not have any branches encompassed within.)

Program	Br. Pred.	Block Predictor	
	BPA	BlPA	BPA
compress	86.99%	87.63%	86.96%
eqntott	94.82%	92.15%	94.64%
espresso	95.36%	91.79%	94.68%
gcc (cc1)	84.94%	81.16%	83.21%
xlisp	92.56%	86.53%	92.40%

BPA - Branch Prediction Accuracy;
BlPA - Block Prediction Accuracy

Table 2: Prediction Accuracies for $f = 12$

4.2.3 Block Termination Cause

Table 3 shows the BTC results for $f = 12$. We have categorized the BTC into 4 categories, as indicated by the last 4 columns of Table 3. From the table, we can deduce that the main reason for the blocks to be terminated (before reaching the fetch size) is due to conditional branches that are predicted to be taken, and cause a control flow out of the block from somewhere in the middle of the block.

Program	4^{th} CB	Call & Return	UB & Jumps	CB Taken
compress	0.00%	6.70%	24.24%	69.10%
eqntott	0.15%	6.60%	7.30%	85.94%
espresso	0.21%	9.65%	5.27%	84.87%
gcc (cc1)	2.97%	20.59%	20.70%	55.74%
xlisp	0.19%	43.78%	15.49%	40.54%

CB - Cond. Branch; UB - Uncond. Branch

Table 3: Block Termination Cause for $f = 12$

4.2.4 Sensitivity Studies

Next, let us vary some of the parameters (that were fixed so far) in order to study their effect on the block predictor's performance. The parameters that are of interest to us in these sensitivity studies are: (i) maximum fetch size (f), (ii) the pattern size (p), and (iii) the number of entries in the BHT (N). Each of these parameters is varied while keeping the rest of the parameters at their default values specified in Section

4.1.4, and the metric used was the effective fetch size (which is our primary metric).

Let us start by considering the effect of f on EFS. Figure 6 plots the EFS obtained for 3 different values of f, namely 8, 12, and 16. As f is increased, EFS also increases. But, the increase is sub-linear, and is likely to taper off, as f is increased further, because of many conditional branches being predicted to be taken.

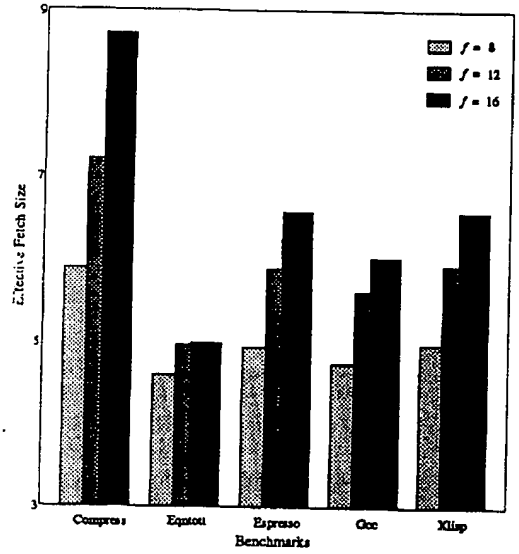


Figure 6: Effect of f on EFS for block predictor

Next, let us study the effect of varying the pattern size p. Figure 7 plots the EFS obtained for the benchmarks for $p = 4$, 6, and 8. It can be seen from the figure that a pattern size of 6 is quite adequate for the block predictor.

Lastly, let us look at the effect of the number of entries in the BHT (N) on the performance of the fetch mechanism. Figure 8 plots both the EFS and the BHT hit rates obtained for the benchmarks for different values of N. Except for gcc, all other benchmarks stand little to benefit from a value of N more than 1024. For programs that require large N, such as gcc, better hit rates can be achieved with smaller N by using set-associative mapping to select the BHT entries.

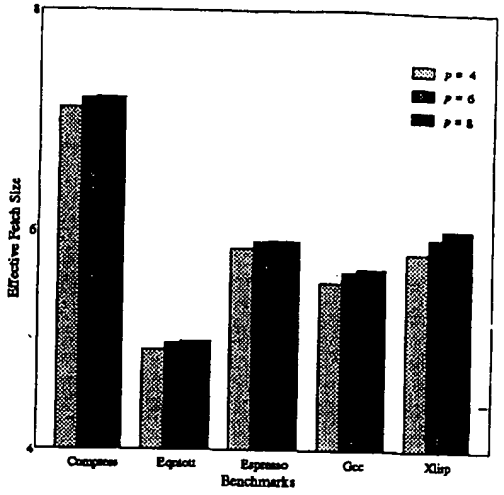


Figure 7: Effect of p on EFS for block predictor

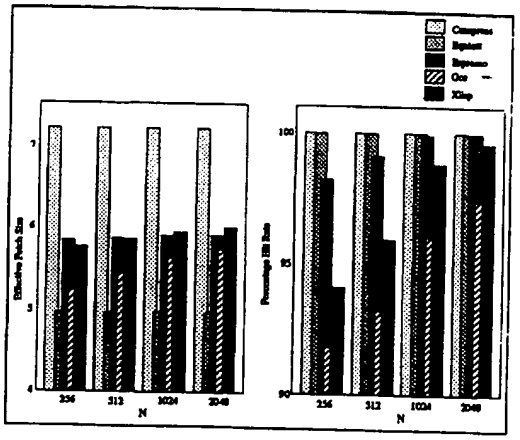


Figure 8: Effect of N on EFS and BHT Hit Rate

5 Conclusions and Future Work

This paper proposed a block-level prediction mechanism for wide-issue superscalar processors. Instead of predicting the outcome of each conditional branch, this scheme predicts the target of an entire (sequential) block of instructions (*i.e.*, it predicts where control is most likely to go after the execution of the

block). This helps to effectively predict the outcome of multiple branches per cycle, thereby enabling the fetch unit and the decoder to bring in more instructions per cycle into the superscalar hardware window. The main advantage of this scheme is the simplicity of the fetch mechanism. The fetch mechanism makes only one prediction per cycle, but could be effectively predicting the outcome of up to 3 branches.

We also conducted a thorough simulation study to verify the potential of this prediction strategy. The results of our experiments with the MIPS architecture help us to conclude that a block-level prediction approach to superscalar processors is profitable, especially when the fetch size is large. It was found to increase the effective fetch size by about 25% over that of branch prediction schemes. Our studies also showed that the main reason for the blocks being terminated prematurely was because the targets returned by the block prediction mechanism quite often forced one of the branches contained within the block to be predicted to be taken.

Future work therefore includes fetching non-straightline code sequences in a cycle so that the fetch mechanism can continue to fetch along the targets of the predicted branches as in [13]. However, we would like to do it without making several individual predictions, to retain the simplicity of our block-level prediction mechanism. Another observation that we made in our experiments was that there were many short branches due to short 'if-then' and 'if-then-else' code sequences. It might be a good idea to encompass these short branches completely within a block, so that those branches do not add to the block's targets.

Acknowledgements

This work was funded by the Department of Electrical and Computer Engineering of Clemson University, and by the NSF Research Initiation Award Grant CCR 9410706.

References

[1] *Byte*, November 1994.

[2] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report TR 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[3] M. C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 93-97, January 1990.

[4] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[5] J.L. Hennessy and D.A. Patterson, *Computer Architecture A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.

[6] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 386-395, 1986.

[7] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[8] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, pp. 6-22, January 1984.

[9] S-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pp. 76-84, 1992.

[10] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, "Control Flow Prediction for Dynamic ILP Processors," *Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO-26)*, pp. 153-163, 1993.

[11] D. N. Pnevmatikatos and G. S. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processors," *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[12] S. Weiss and J.E. Smith, *POWER and PowerPC*. San Francisco: Morgan Kaufmann, 1994.

[13] T-Y Yeh, D. T. Marr, and Y. N. Patt, "Increasing Instruction Fetch Rate via Multiple Branch Predictions and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 67-76, July 1993.

[14] T-Y Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.

Results 1 - 1 of 1

1 Multiple-block ahead branch predictors

André Seznec, Stéphan Jourdan, Pascal Sainrat, Pierre Michaud

October 1996 Proceedings of the seventh international conference on Architectural su operating systems, Volume 30 , 31 Issue 5 , 9

Full text available: pdf(1.30 MB) Additional Information: full citation, abstract, references, citing

A basic rule in computer architecture is that a processor cannot execute an applic instructions. This paper presents a novel cost-effective mechanism called the two Information from the current instruction block is not used for predicting the addre rather for predicting the block following the next instruction block.This approach c bottle-neck exhibited by wide-dispatc ...

Results 1 - 1 of 1

Multiple-Block Ahead Branch Predictors

André Seznec † Stéphan Jourdan ‡ Pascal Sainrat ‡ Pierre Michaud †*

IRISA †
Campus de Beaulieu
35042 Rennes, France
{seznec,pmichaud}@irisa.fr

IRIT ‡
Université Paul Sabatier
31062 Toulouse, France
{jourdan,sainrat}@irit.fr

Abstract

A basic rule in computer architecture is that a processor cannot execute an application faster than it fetches its instructions. This paper presents a novel cost-effective mechanism called the two-block ahead branch predictor. Information from the current instruction block is not used for predicting the address of the next instruction block, but rather for predicting the block following the next instruction block.

This approach overcomes the instruction fetch bottleneck exhibited by wide-dispatch "brainiac" processors by enabling them to efficiently predict addresses of two instruction blocks in a single cycle. Furthermore, pipelining the branch prediction process can also be done by means of our predictor for "speed demon" processors to achieve higher clock rate or to improve the prediction accuracy by means of bigger prediction structures.

Moreover, and unlike the previously-proposed multiple predictor schemes, multiple-block ahead branch predictors can use any of the branch prediction schemes to perform the very accurate predictions required to achieve high-performance on superscalar processors.

1 Introduction

Two different approaches are used in current processors to achieve high performance: "brainiacs vs. speed demons" [6]. While "brainiacs" favor the parallel execution of instructions and "speed-demons" favor a high clock rate, both approaches are facing a similar difficulty with fetching instructions at a sufficient rate. The purpose of this paper is to propose a new branch prediction mechanism allowing to increase the instruction fetch rate for both approaches.

"Brainiac" processors To best exploit the available ILP, "brainiac" processors are using a large number of functional units working in parallel.

Unfortunately, the instruction-fetch mechanisms implemented in current commercial microprocessors do not fully exploit the potential parallelism. For these processors, the instructions fetched in a single cycle most often belong to the same basic block, and are not usually permitted to span two cache lines. Since a processor cannot execute instructions faster than it fetches them, these constraints significantly impair performance, particularly on codes featuring many small basic blocks.

A partial solution to the instruction fetch bottleneck, is to fetch instructions belonging to multiple consecutive basic blocks, as is done in processors such as the POWER2 [18]. To solve the whole problem, multiple non-consecutive basic blocks must be fetched in a single cycle as most basic blocks are only five instructions long. Indeed, the potential parallelism has been shown to be higher than six instructions per cycle in general-purpose integer applications while assuming a perfect instruction-fetch mechanism [11]. A processor featuring such a mechanism would have to predict multiple targets and branch outcomes in a single cycle.

In superscalar processors, blocks of consecutive instructions are fetched in parallel. The last instruction of such a block is either a branch or is determined by some implementation constraints (for instance, the boundary of a cache block or the maximum number of instructions in the block). Throughout this paper, we refer to processors that can fetch only one basic block per cycle as *single I-fetch processors*, to processors that can fetch two non-consecutive blocks per cycle as *double I-fetch processors*, and to *multiple I-fetch processors* as an extension to the latter case.

We show in this paper that double I-fetch processors will have a major performance advantage over single I-fetch processors for a dispatch width of six or higher. Our belief is that future generations of "brainiac" processors will be multiple I-fetch processors.

"Speed-demon" processors To achieve high performance, "speed demon" processors rely on moderate numbers of functional units, but a very high clock rate. In such processors, a single instruction block is dispatched in each cycle, but the branch predictor is often a critical path in the processor. In current microprocessors, either the branch prediction and the address generation are completed in a single cycle, or pipeline bubbles are inserted on each predicted taken branch (e.g. on DEC 21164 [5], PentiumPro [8] or MIPS R10000 [13]), therefore potentially limiting the performance. Another way to deal with this critical path is to reduce the number of entries of the one-cycle access prediction structures (e.g. HP PA-8000 [7]), thus impairing the prediction accuracy.

*This work was partially supported by PRC-GDR AMN (CNRS)

ASPLOS VII 10/96 MA, USA
© 1996 ACM 0-89791-767-7/96/0010...$3.50

In this paper, we will show that pipelining the branch prediction is possible on single I-fetch processors, and that branch prediction therefore need not be a critical path on such processors.

The two-block ahead branch predictor In conventional branch prediction mechanisms, information associated with the current instruction block such as its memory address, is used to predict the next instruction block. Previous multiple predictors [19, 4] also rely on a single piece information to predict the two subsequent instruction blocks.

In this paper, we propose a complete and cost-effective mechanism called the *Two-Block Ahead Branch Predictor*. The originality of our mechanism is to use information associated with the current instruction block to predict the block following the next instruction block. Such an approach can obviously be extended to predict blocks with even further advance; we refer to this as a *Multiple-Block Ahead Branch Predictor*.

The *Two-Block Ahead Branch Prediction* consists of a two-block ahead branch prediction table, a two-block ahead branch target buffer and a two-block ahead return stack.

Most previously proposed prediction schemes can be adapted to the *Two-Block Ahead Branch Predictor* leading to high prediction accuracy. Moreover, the amount of information stored in the two-block ahead branch predictor is not higher than in a conventional branch prediction mechanism.

The two-block ahead predictor can be used in a double I-fetch processor: both fetch addresses are used to predict the two subsequent instruction blocks to fetch on the next cycle. Implementing the two-block ahead branch predictor in a single I-fetch "speed demon" processor allows the branch prediction to be pipelined.

Paper organization The remainder of the paper is organized as follows. Related work is discussed in section 2. Section 3 compares single I-fetch and multiple I-fetch processors. Section 4 introduces the two-block ahead branch predictor and presents its implementation for a double I-fetch processor. Section 5 shows that the branch prediction process can be pipelined in single I-fetch "speed demon" processors by means of our two-block ahead branch predictor. Finally, simulation results are reported in section 6. Section 7 concludes the paper.

2 Related Work

To our knowledge the pipelining of the branch prediction has never previously been addressed.

Only a few studies [19, 4, 3] have addressed the problem of fetching multiple non-consecutive basic blocks in a single cycle.

Yeh, Marr, and Patt To fetch two basic blocks in a single cycle, Yeh et al. [19] proposed storing 6 addresses in each entry of their *Branch Address Cache* (BAC): T, N, TT, TN, NT, and NN, where N and T refer to the outcome of the branches (*not-taken* and *taken* respectively). The branch prediction mechanism can predict two branches in a single cycle. According to the prediction made by a history-only based scheme (address-based schemes give lower prediction accuracy since they use the same address for both predictions), the addresses of the two subsequent basic blocks are returned with a hit in the BAC. When a branch is resolved for the first time, an entry is allocated in the BAC with fields T and N set (primary fields). If the previous fetch address had a valid primary branch entry in the BAC with secondary fields cleared, and if there was enough bandwidth to fetch another basic block, then T and N are also inserted in these secondary fields. This introduces wasted fields when entries are allocated for primary branches with not enough fetch bandwidth left for a second basic block (with a dual-ported instruction cache, this occurs each time a basic block lies across an aligned block boundary). Furthermore, since most of the branches are unidirectional, two third of the fields are under-utilized.Finally, a basic block can belong to several BAC entries, so their mechanism does not require any static partitioning. Redundancies are however created, but the scheme does not rely on any compiler work.

Dutta and Franklin In [14], the authors proposed splitting the *Control Flow Graph* (CFG) into subgraphs. To fetch two non-consecutive basic blocks even when they belong to different cache lines, they use tree-like subgraphs of depth 3 [4]. Nodes of the subgraphs are straightline pieces of code (basic blocks). Intermediate nodes (depth 0 and 1) can be terminated by any control-changing instructions while the last nodes (depth 2) are terminated by single-target instructions (either unconditional branches, procedure calls, or non-branch instructions), and their lengths are limited by the instruction-fetch bandwidth. Intermediate outcomes are not predicted. Instead, one path is predicted in the subgraph among four. All parameters required to describe a subgraph are stored in a Subgraph History Table (SHT). Except for the prediction mechanism, this method is much like Yeh's approach. To avoid redundant information in the SHT, each basic block should belong to only one subgraph. Their scheme mostly relies on compiler work to partition the CFG into tree-like subgraphs of depth 3. One should note that basic block duplication implies that history information is now shared out. Hence, redundancy impairs more significantly performance than in Yeh's approach. Furthermore, since each entry in the SHT holds a rigid subgraph structure, there might be many under-utilized or wasted fields. Indeed, a static CFG is not as simple as a tree and it cannot be perfectly partitioned into tree-like structures. With optimized code, the prediction mechanism gives good accuracy results compared to non-hybrid schemes without a lot of additional logic.

Conte, Menezes, Mills, and Patel In [3], the authors introduced a mechanism called the *Collapsing Buffer* that achieved *merging* [10]. This mechanism can fetch multiple basic blocks in a single cycle as long as they belong to the same cache line, and otherwise performs some alignments between two basic blocks by means of a pipelined fetch mechanism (*banked sequential*). The mean instruction-fetch throughput is at most one cache line with no restriction on the number of predicted branches within the cache line. The scheme features an interleaved coupled BTB/BHT providing one entry to each instruction of a cache line. The Collapsing Buffer scheme is efficient as long as branch targets address the same cache line and performs well on their execution model retiring less than 2.5 instructions per

cycle on integer applications. Another major drawback is the requisite use of an address-only based prediction scheme. Moreover, as the I-cache line size keeps growing in current processors, the interleaving factor of the BTB grows as well and the collapsing logic becomes more complex. Although this approach is interesting, our purpose is to go one step beyond by fetching multiple basic blocks in a single cycle, even when they belong to different cache lines. One should note that our approach is not incompatible with collapsing.

Generic comment The first two methods create several layers of BTB: BTB1, BTB2 where BTB1 makes a prediction one branch ahead and BTB2 makes a prediction two branches ahead (it happens that both combine BTB1 and BTB2 into the same structure, *joining* on the search key). Our elegant approach only uses a single kind of BTB, BTB2, and dual-ports it to achieve two predictions and address computations in a single cycle. Hence, there is no difference between the branch prediction numbers in terms of the first and second prediction.

It should be noted that none of these previously published solutions can be extended easily to pipeline the branch prediction.

3 Single vs. Multiple I-fetch Processors

While multiple basic block fetch mechanisms have already been proposed, there was no clear study showing that such mechanisms would provide performance enhancements. The purpose of this section is to show that despite data dependencies and resources hazards, multiple fetch mechanisms would give significant performance improvement in wide-dispatch out-of-order processors.

Traces The experimental results presented in this paper are based on the programs from the SPEC92 suite [17]. Programs from both the CINT92 and the CFP92 collections are considered in the presented evaluations. The benchmarks were compiled on a R4600-based SGI workstation using cc and the standard makefiles provided with the suite (with all optimizations turned on). We used the PIXIE profiler [16] to collect instruction traces from a real processing of the SPEC benchmarks, including library calls. These traces fed our simulator.

Due to time constraints, the smallest input files or slightly modified versions have been used in order to run the programs to completion. In all, more than 600 million instructions have been captured (with all NOPs removed from the traces).

Machine Model The modeled architecture depicted in figure 1, implements out-of-order and speculative execution policies in order to best exploit ILP. In brief, after being fetched, instructions are decoded and dispatched in-order from the instruction-dispatch buffer to the instruction-issue buffer. The upper bound of the number of instructions dispatched each cycle defines the dispatch width. Registers are renamed using a map table during the dispatch process. Instructions in the issue buffer may be issued out-of-order when all their operands are available, and a *max-dependent* selection mechanism as described in [1] is used when more than one instruction compete for the same functional unit access. To enforce precise interrupt management, a history buffer similar to the active list of the MIPS R10000,

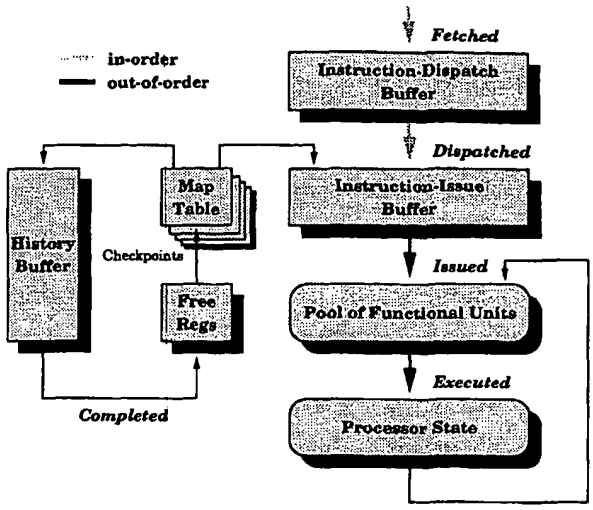


Figure 1: Simulated Out-of-Order Microarchitecture.

Parameters	DW4	DW6	DW8
Dispatch Width	4	6	8
Lookahead Window Size	32	64	96
Issue Buffer Depth	28	48	72
Fixed-Point Units	3	4	5
Floating-Point Units	2	2	2
Branch Units	2	2	3
Data-Cache Ports	2	3	4

Table 1: Model Configurations.

records the previous mappings discarded by the renaming process during the dispatch stage. Checkpoints of the map table (architectural state) are established at every branch in order to recover from branch misprediction in one cycle, regardless of the number of mapping modifications recorded in the history buffer. With such a scheme, the history buffer is only used to recover from other exceptions and to keep track of the state of the physical registers in order to free them when the instructions complete. When an instruction finishes execution, its result updates the processor state but its corresponding entry remains in the history buffer until all previously dispatched instructions can no longer produce interrupts. Instructions capable of generating interrupts are conditional and indirect branches (mispredictions), divides, and memory accesses. In the latter instruction class, subsequent entries may be dequeued as soon as the address is computed. Each cycle, multiple out-of-order instruction retirements can be made, freeing the physical registers to be reused in the renaming process.

A previous study [11] has shown that configurations reported in table 1 of such out-of-order architectures give almost no performance loss over perfect configurations only limited by the size of the lookahead window, assuming an ideal-fetch mechanism and no misprediction. The instruction latencies used in the simulations were those of the PowerPC 604 [9]. The mean-IPC val-

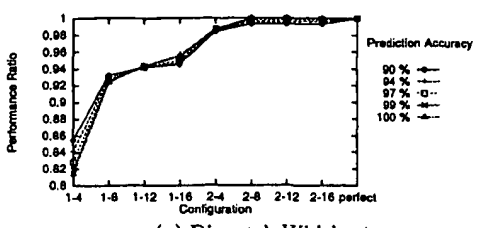
ues varied from 3.6 to 6.5 on integer programs according to the dispatch width (4, 6, and 8 instructions dispatched per cycle). We keep their configurations (DW 4, DW 6 and DW 8) in order to evaluate the fetch mechanisms. The lookahead window is the maximum number of dispatched instructions that can be processed at the same time, sometimes referred as the instruction window. Moreover, we assumed for all the models a unified issue-buffer, and a maximum number of 16 checkpoints. Such a value does not degrade the performance of any of the models.

In most processors, the fetch mechanism consists mainly of three parts: an instruction cache from where the instructions are fetched, an instruction-dispatch buffer where the instructions are maintained waiting to be dispatched, and some branch prediction structures predicting the outcome and the target address of any fetched branch. The dispatch buffer decouples the instruction fetching from the dispatch process, sustaining a better throughput in the presence of cycles in which only a small number of instructions can be fetched (the buffer can be filled in a single cycle). Perfect instruction and data caches are used throughout this section.

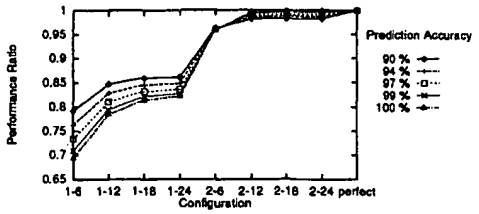
Multiple I-fetch is useless with long basic blocks The CFP92 subset features long basic blocks close to fifteen instructions long on average. Simulations on floating-point programs, not reported here, have shown that a single I-fetch processor is effective whatever the dispatch width may be provided a deep dispatch buffer. A 8-wide single I-fetch processor gives over 97.3 % of the perfect performance when the prediction accuracy is higher than 90 %.

Double I-fetch is useful with small basic blocks The average size of the basic blocks in the CINT92 subset is five instructions long. Figures 2 (a) through (d) show the results on integer programs according to the depth of the instruction-dispatch buffer, the prediction accuracy, and the dispatch width. Only the geometric means of the results are reported. Configurations are based both on the number of basic blocks fetched in a single cycle, and on the depth of the dispatch buffer. For instance, configuration 2-8 denotes a double I-fetch processor featuring a 8-deep dispatch buffer.

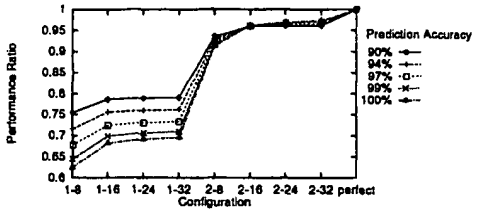
In (a), (b), and (c), performance ratio is the relative performance (IPC) between the evaluated configuration and a configuration featuring a perfect fetch mechanism, the prediction accuracy remaining the same. The curves clearly state that 4-wide processors do not require any improvement over a single I-fetch policy except for a 8-deep dispatch buffer, giving 93 % of the perfect performance whatever the prediction accuracy may be. On the other hand, 6-wide and especially 8-wide double I-fetch processors give a huge improvement of performance over single I-fetch processors. In a 8-wide processor, the improvement is between 20 and 40 % depending on the prediction accuracy, assuming a deep dispatch buffer. Moreover these results bring to light that fetching more than two basic blocks in a single cycle is not effective for an 8-wide machine (while keeping binary compatibility) as double I-fetch mechanisms provide nearly 100 % of performance. One should note that the relative benefit when fetching two basic blocks in a single cycle increases with the branch prediction accuracy. Finally, an instruction buffer twice as big as the dispatch width is required in any case.



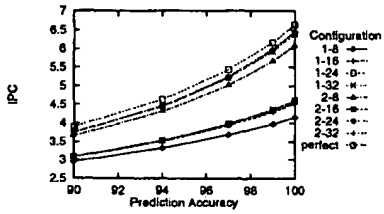
(a) Dispatch Width: 4



(b) Dispatch Width: 6



(c) Dispatch Width: 8



(d) Dispatch Width: 8

Figure 2: Performance of Single and Multiple I-Fetch Processors.

As shown in figure (d) for a 8-wide processor, it is more effective to increase the number of blocks fetched per cycle than to improve the prediction accuracy. Nevertheless, these two optimizations are not exclusive and they each give new opportunities for performance improvement.

4 The Two-Block Ahead Branch Predictor

This section is illustrated with the implementation of a Two-Block Ahead Branch Predictor for a double I-fetch processor (figure 4). The implementation of a pipelined Two-Block Ahead Branch Predictor for a single I-fetch processor will be detailed in the next section.

As stated in the introduction, the two-block ahead branch predictor uses information associated with the current instruction block to predict the address of the instruction block that is two blocks ahead. Its principle is illustrated in Figure 3 where **Ai**, **Bi**, **Ci**, and **Di** are the basic block starting addresses and **Aa**, **Bb**, **Cc**, **Dd** are the branch addresses. Any of the control-flow transitions can be fall-through. While the instruction blocks **A** and **B** are fetched, the two-block ahead branch pre-

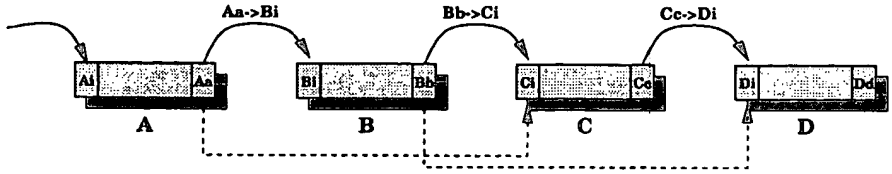
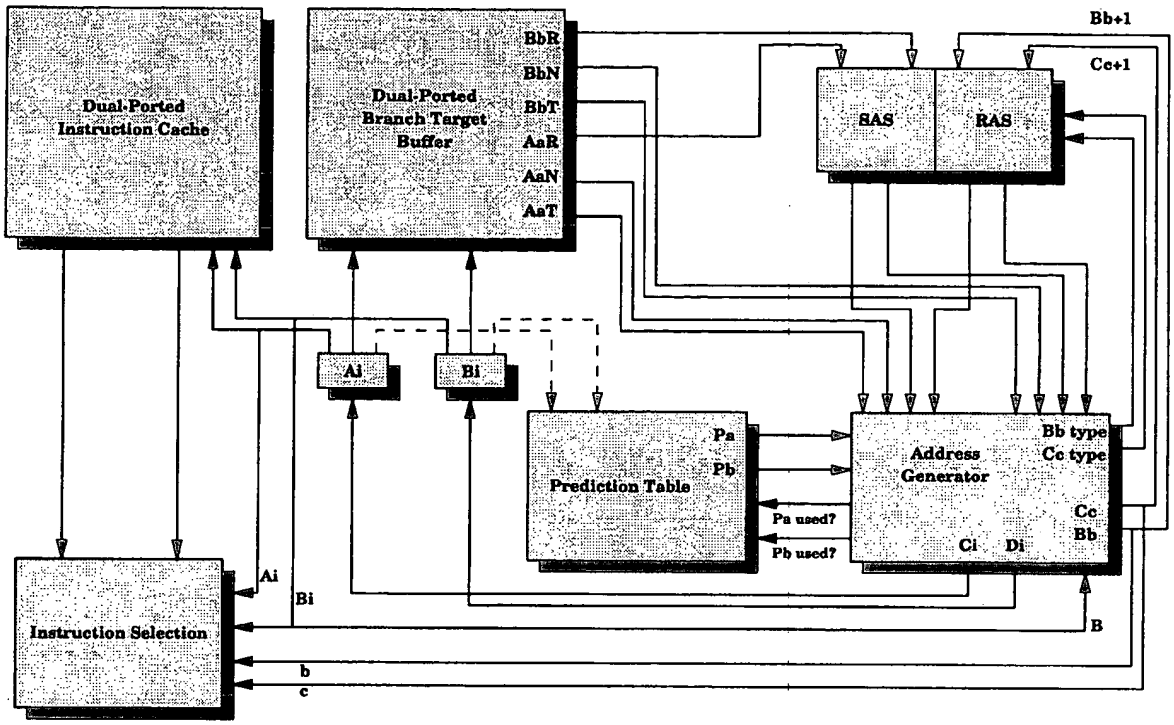
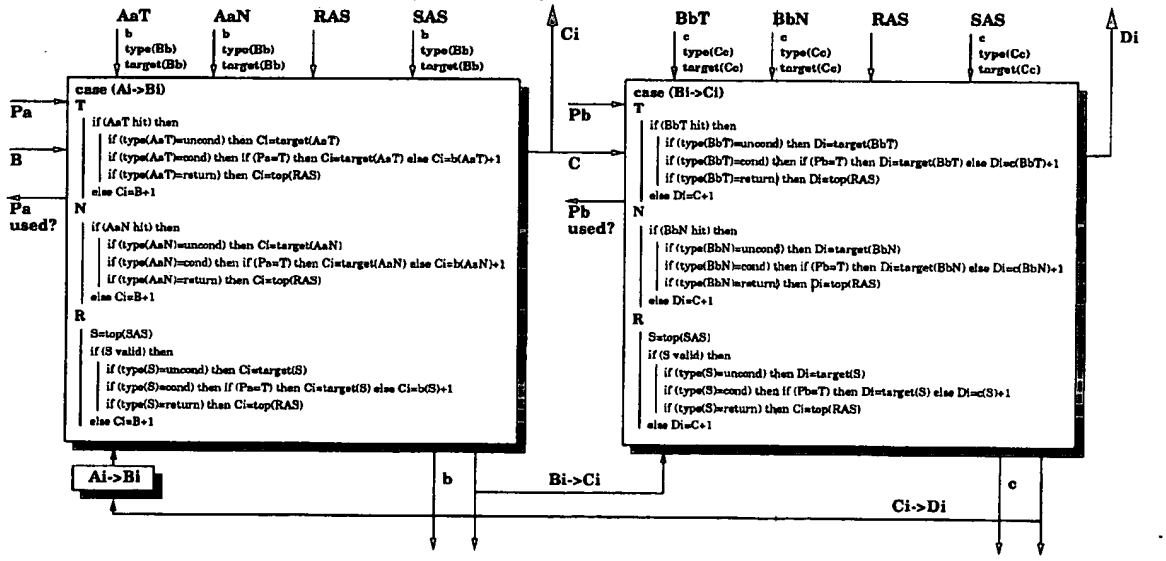


Figure 3: Two-Block Ahead Branch Information.



(a) Predicting Two Basic Blocks



(b) Instruction Address Generation

Figure 4: Two-Block Ahead Branch Predictor in Double I-Fetch Processor.

dictor uses **Aa** instead of **Bb** to predict **Ci**, and **Bb** to predict **Di**. The behavior of each part of the predictor is described in the following sections.

4.1 The Two-Block Ahead Branch Prediction Table

Instead of using the address and the history register of the conditional branch (**Bb**,**Hb**) to predict its outcome **Ci**, our scheme always uses the address and the history register of the previous branch (**Aa**,**Ha**) to predict **Ci**. Such a scheme can be adapted to use any branch prediction schemes combining address and history (see for instance [12]).

We will show in the section 6 that the prediction is as accurate as if the address of the branch had been used instead. In figure 4, **Pa** and **Pb** refer to the predicted outcome when the Branch Prediction Table (PT) is indexed with (**Aa**,**Ha**) and (**Bb**,**Hb**) respectively.

4.2 The Two-Block Ahead Branch Target Buffer

BTB entry description The two-block ahead BTB records information for a given branch in an entry associated with both the address of the previously fetched block and the type of transition between both blocks.

When a taken branch **Bb** is mispredicted or misfetched, a BTB entry is allocated to record its target (**Ci**), type (conditional, unconditional, indirect, or return) i.e. 2 bits, and position **b** in block **B** (for instance, 4 bits for a 16-instruction cache-line size).

Unlike conventional designs, the BTB entry is not tagged with the address **Bb**, the starting address **Bi** of the instruction block containing **Bb**, or the address of the cache line containing **B**. But it is associated with the address of **Aa**, the last instruction in the previous instruction block. The BTB is indexed with (1) the address of **Aa**, and (2) the type of the transition between **Aa** and **Bi** (**Aa**→**Bi**) (2 bits). If no branch was fetched with **A**, **a** would be the last instruction in block **A**. There are three types of transition: **T** (**Aa** is a non-return taken branch), **N** (**Aa** is a non-taken conditional branch or a non-branch instruction), and **R** (**Aa** is a call). Type **R** is special and is further explained when we introduce the procedure return mechanism.

BTB read We illustrate here the read of the two-block ahead branch target buffer on a double I-fetch processor. Let us detail the information that is available at the beginning of the cycle.

- both addresses **Ai** and **Bi** are available; the block addresses **A** and **B** are used to access the BTB in addition to the I-cache.
- the branch position **a** and the transition type **X** between **Aa** and **Bi** were determined during the previous cycle.

All the information required to compute **Ci** is known. We can therefore check for BTB entry **AaX** to compute **Ci**. This entry would hold the target **Ci** of branch **Bi**, its type and its position. If **Bb** is a conditional branch, the outcome is provided by the two-block ahead branch prediction table. The whole process for computing **Ci** is detailed in figure 4.

Some pieces of information needed for computing **Di** are not directly available at the beginning of the cycle: position **b** of the branch in block **B** and transition type **Y** from **Bb** to **Ci**. This information is obtained on the fly with **Ci**:

- if **AaX** hits in the BTB then this information is part of the **AaX** entry.
- if **AaX** misses in the BTB then we assume that no branch in **B**, and **b** would be the last instruction in line **B** and transition **Y** is assumed to be fall-through.

Once these values are produced, the tag checking for the BTB entry **BbY** may begin and the address **Di** is computed in a similar way as **Ci**.

To enable parallel read of both entries **AaX** and **BbY**, any entry **EeZ** is mapped in the BTB as follows: low-order bits of **E** are used to address the set (BTB indexing), and both **eZ** and high-order bits of **E** are used to tag the entry allocated within the set.

So indexing the BTB with **A** and **B** may be done in parallel, but the tag-matching for **AaX** and **BbY** is partially serialized. This constraint is part of the parallelism vs. speed tradeoff ("brainiacs vs. speed demons"). However, such a process can be easily pipelined within the fetch stage, further featuring the structure update process.

Storage cost The amount of information stored in a two-block ahead BTB entry is only a few bits wider than in a conventional BTB (position **b** of the branch in **B** and the transition type **Aa**→**Bi**).

On the other hand, two entries may be associated with a single address **Aa**. The BTB entry **AaT** records information for a branch in the target basic block of branch **Aa**, and entry **AaN** records information for a branch in the fall-through basic block of branch **Aa**.

Some redundancies may be created when a branch **Bb** has more than one predecessor block: in this case, a target may be represented several times in the BTB. However, our simulations show that the two-block ahead BTB does not require many additional entries to achieve the same hit ratio as a conventional BTB.

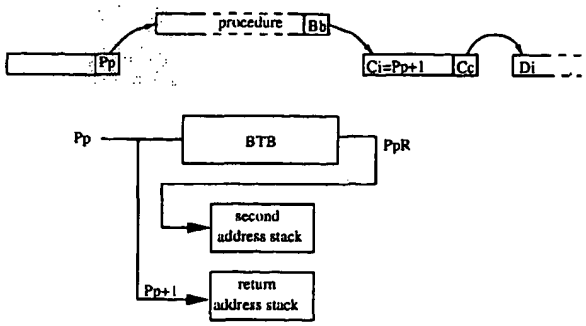
Associativity Since most of the conditional branches are either mostly taken or mostly fall-through [20], the BTB will often record only one of **AaT** or **AaN**. Thus the associativity required in the two-block ahead BTB will not be much higher than in a conventionnal BTB.

4.3 Coping with Procedure Returns

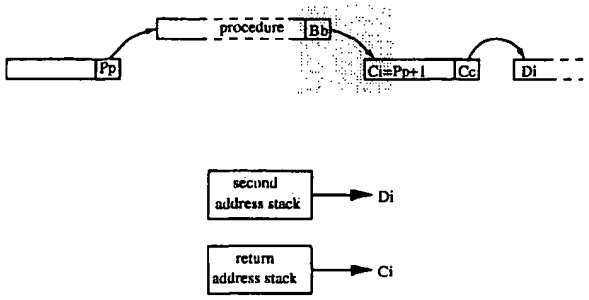
Procedure-return jumps are a special case where using a BTB alone is inefficient: the target address may change very often. To cope with this difficulty, many recent processors implement a Return Address Stack into which return addresses are pushed when the calls are fetched. The two-block ahead branch predictor may also use a Return Address Stack for predicting return addresses.

Nevertheless, when using two-block ahead branch prediction, the address **Bb** of the return branch should be used to predict the instruction block **Di** following the return target block (figure 5.a). **Di** is intuitively more dependent on the return target **Ci** which may vary frequently for the same return branch than on the return branch **Bb** itself. Then predicting this block with the two-block ahead branch target buffer presented above is likely to result in many misfetches. Yeh et al. reached the same conclusion and proposed the fetching of only a single block in this case [19].

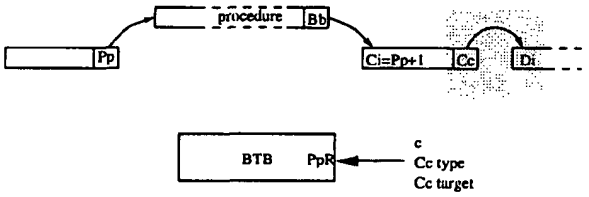
A specific solution for coping with predicting the instruction block following the return target block is presented here.



(a) - Call Pp is detected at fetch time, the stacks are written



(b) - Ret Bb is detected at fetch time, the stacks are read



(c) - Branch Cc was mispredicted, the BTB is updated

Figure 5: The two-block ahead return stack.

Second Address Stack As already mentioned, the address **Di** of the instruction block following the return target is more dependent on the address of the return target **Ci** than on the return address **Bb** itself. A specific difficulty is that the return target is unknown when its block successor has to be predicted.

Nevertheless, a branch strongly associated with the return target **Ci** has been already issued in the instruction flow: the procedure call **Pp= Ci-1**. We propose associating the information on the instruction block **Di** with a BTB entry associated with the procedure call **Pp**.

This is illustrated in figure 5. A special entry type **R** is introduced in the Two-Block Ahead BTB for dealing with this case.

A BTB entry PpR is allocated instead of an entry BbT when a branch in block C is mispredicted to keep information about the branch Cc (figure 5 (c)). It should be noted that information Pp can be easily recovered because Pp equals Ci-1.

- A BTB entry **PpR** is allocated instead of an entry **BbT** when a branch in block **C** is mispredicted to keep information about the branch **Cc** (figure 5 (c)). One should note that information **Pp** can be easily derived since **Pp** equals **Ci-1**.
- The BTB is searched for an entry **PpR** whenever a call instruction **Pp** is fetched (figure 5 (a)). This information must be kept until the return instruction **Bb** is fetched. For this purpose, the two-block ahead branch predictor uses a Second Address Stack (SAS). A copy of the **PpR** entry is pushed into the SAS whenever **PpR** hits in the BTB. Otherwise, an invalid entry is pushed. Notice that the same number of entries are pushed in the return Address Stack and in the SAS.
- When a return is popped from the Return Address Stack, an entry is popped from the SAS to accurately predict a branch in the subsequent basic block if any (figure 5 (b)). Notice that this branch may be a return.

The whole process is further detailed in figure 4.

4.4 A further optimization

When using the presented two-block ahead branch predictor, and when a branch **Bb** is predicted not taken, the following predicted instruction block begins at address **Bb+1**.

But **Bb** may not be the last instruction in the block which was read in parallel from the I-cache. For instance, in the example illustrated in figure 6 (a), three consecutive fetches are issued on the same cache block. Instruction blocks E, F and G are read in parallel on the first fetch, but blocks F and G are then discarded.

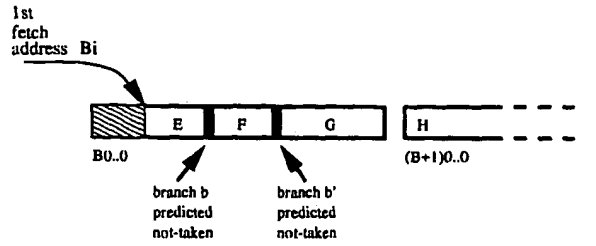
Such a situation wastes I-cache bandwidth. Being able to pick at the same time all the useful consecutive instructions read in parallel (i.e. instruction blocks E,F and G in the illustrated example) in an instruction cache block would obviously save many fetch cycles.

A complex general case Several consecutive conditional branches may lie in the block **Bi** of instructions read in parallel. Ideally, the instruction fetch mechanism should be able to forward the entire sequence of consecutive useful instructions in this block for further processing , then bypass all the consecutive not-taken branches (figure 6 (b)).

However, predicting the instruction block fetched after this sequence is a rather challenging problem when using information associated only with the predecessor instruction block **Aa**:

- The fetched block may be any of the targets of the consecutive conditional branches or it may also be the fall-through block.
- A branch prediction must be performed for each one of the conditional branches in the cache block.

We have not yet been able to find a simple solution for this general case although collapsing may lead to a solution.



(a) worst case

	fetch address	fetched instruction groups
1st fetch	Bi	E
2nd fetch	Bb+1	F
3rd fetch	Bb'+1	G
4th fetch	(B+1)0..0	H

(b) ideal case

	fetch address	fetched instruction groups
1st fetch	Bi	EFG
2nd fetch	(B+1)0..0	H

(c) branch b' bypassed

	fetch address	fetched instruction groups
1st fetch	Bi	E
2nd fetch	Bb+1	FG
3rd fetch	(B+1)0..0	H

Figure 6: Bypassing not-taken branches.

Bypassing the last branch in the instruction block When **Bb** is the last branch in the fetched instruction block and is not-taken, then the successor block of **Bb** is the block beginning at address **(B+1)**0..0.

Let us suppose that the information **L** "the branch is the last one in the cache block (or not)" is recorded in the BTB entry. Then the instruction-address generator can use this information to compute the predicted instruction block as follows. When **L** is set and the branch **Bb** is predicted not-taken, the predicted instruction block is the block beginning at at address **(B+1)**0..0 instead of the block beginning at address **Bb+1** which is fetched during this cycle.

The extra hardware required for implementing this optimization is quite low: an extra bit in each BTB entry and some logic in the decode stage for computing **L**. On the other hand, it systematically saves one instruction block fetch when it applies (figure 6 (c)). It should be noted that this solution may also be adapted to conventional branch target buffers, and that its efficiency may be highly improved with software ordering of most likely taken branches to be fall-through.

4.5 Two-Block Ahead Branch Predictor in a double I-fetch processor

When using a double I-fetch processor, the I-cache must either be fully double-ported or interleaved [18].

When using the two-block ahead branch predictor, the branch prediction table and the branch target buffer must also be either fully double-ported or interleaved.

When the I-cache is interleaved, the branch prediction table and the branch target buffer may be also interleaved in the same way. That is when blocks **A** and **B** are conflicting on the I-cache, they are also conflicting on the branch prediction table and the branch target buffer (and vice-versa). In this case, using an interleaved two-block ahead branch predictor will not impair performance at all.

Furthermore, the RAS and SAS must be able to deliver two addresses per cycle:

- When **Bb→Ci** is a return, the return stack must deliver **Ci** and the transition **Cc→Di**. In this case, the SAS is used to compute **Di**.

  When transition **C→Di** is also a return, a second read is done on the return stack.
- When **Aa→Bi** is a return, the SAS is used to compute **Ci**. When **Bb→Ci** is also a return, a second read of the SAS is used to compute **Di**.

When both transitions **Bb→Ci** and **Cc→Di** are calls, these two stacks have also to be able to accept two pushes per cycles.

5 Single I-Fetch Processors and Two-Block Ahead Branch Prediction

In Section 6, we will show that branch prediction information can be associated with the previous branch instructions without degrading the prediction accuracy. With such a scheme, two addresses are predicted in a single cycle in double I-fetch "brainiac" processors as shown in the previous section. Instead of exploiting more parallelism, another way to get performance improvement is to increase the clock rate, leading to single I-fetch "speed-demon" processors. In this section, we first show that the instruction-address generator stage may be the critical path of the processor, then we show that pipelining the instruction-address generation process (figure 7) in such single I-fetch processor can be done by means of the two-block ahead branch predictor.

5.1 Instruction Address Generation may be a Critical Path

In current single I-fetch processors, both the I-cache and the branch predictor are accessed with the current instruction block starting address. By the end of the cycle, the starting address of the next instruction block must be generated. In some of the processors, the I-cache access time is longer than the cycle time. For instance, the Intel PentiumPro features a pipelined I-cache access completed within two cycles.

As far as the current instruction block address is used to predict the next instruction block, either the instruction address generator can compute the starting address of the next instruction block in a single cycle, or bubbles are inserted in the pipeline in the case of branches as in the Intel PentiumPro. Indeed, accessing the prediction structures in the PentiumPro is spread over two cycles, mainly because its structures feature a high number of entries. Reducing the number of entries impairs the performance, especially on such an heavy-pipelined processor. The instruction-address generation process is quite complex because it includes several consecutive steps:

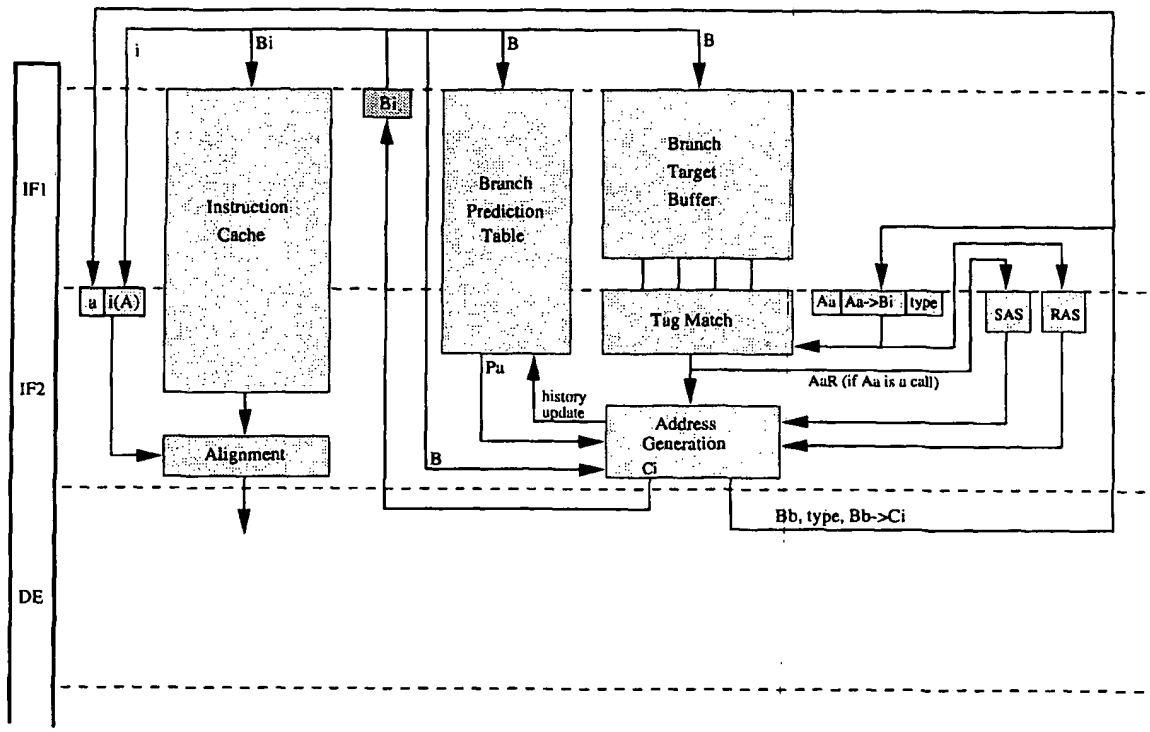


Figure 8: The Two-Stage Pipelined Branch Predictor (cycle t+1).

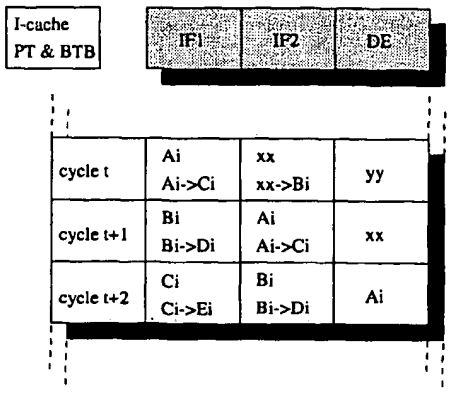


Figure 7: Pipelined Address Generation Timings.

1. Parallel accesses to the BTB, the Prediction Table (PT), and the return address stack. Computation of the fall-through address.
2. Prediction of the outcome and selection of the generated address. Possible updates of the return stack and the branch history register.

In particular, the read of a set-associative BTB featuring a high number of entries is time consuming. Achieving the complete instruction address generation process in a single cycle may be a challenging problem in high clock-speed processors. The instruction address generator might then be the critical electrical path, determining the processor cycle time.

5.2 Pipelining the Instruction Address Generation Process

A two-stage pipelined branch predictor is depicted in figure 8. The BTB and PT illustrated in this figure are two-block ahead BTB and PT implementations computing only one address in a single cycle.

Let **Ai**, **Bi**, and **Ci** be the starting addresses of the instruction blocks respectively fetched at cycle t, t+1, and t+2, **Aa**, **Bb**, and **Cc** be the addresses of the last instruction in these blocks, **A**, **B**, and **C** be these block numbers, **Ha**, **Hb**, and **Hc** be the branch history registers during cycle t, t+1, and t+2. The behavior of the pipelined address generator is represented in figure 7 and is as follows:

1. **cycle t**: In first stage IF1 of the pipeline, **A** and **Ha** are used to index the PT and the BTB. The fetching of the instruction block **A** begins in the instruction cache. At the end of the cycle, **Bi** and **a** flow out from the second stage IF2 as does the

type of the transition from **Aa** to **Bi** (**T**, **N**, or **R**).

2. **cycle t+1**: In stage IF1, **B** and **Hb** are used to index the PT and the BTB while the fetching of **B** begins in the instruction cache. In stage IF2, the access to the instruction cache, and to the BTB and the PT with **A** and **Ha** are completed. In particular, the tag check on the BTB is performed during this cycle.

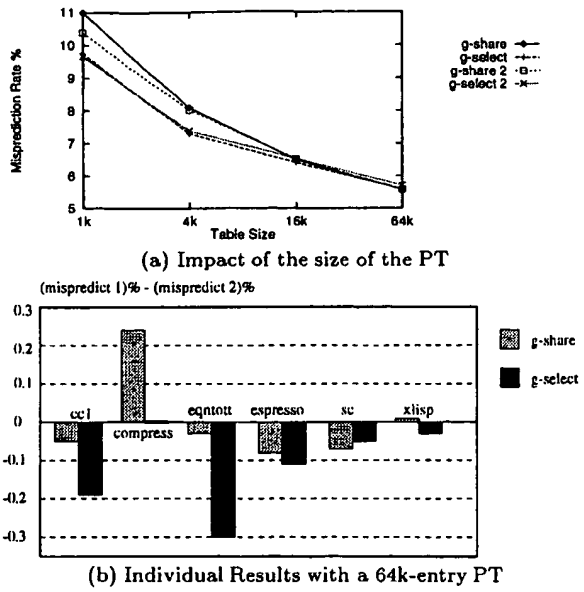
   The fall-through address for block **B** is also computed (**Bb+1**). Depending on the type of the transition from **Aa** to **Bi**, and on the information flowing out from the BTB and the PT, **Ci** is chosen among four addresses (the target addresses flowing out from the BTB, top of the RAS or top of the SAS, or the fall-through address **Bb+1**) as related in the first algorithm of figure 4. By the end of cycle t+1, position b and transition type **Bb→Ci** are forwarded to proceed the tag-matching process on the next cycle.

3. **cycle t+2**:**C** is used to index the instruction cache and to compute the two-block ahead instruction block starting address in stage IF1. The instruction block **C** is decoded.

One should note that some information (the type of the transition from **Aa** to **Bi**, the position **a** of the last instruction in block **Ai**) produced during cycle t are used during cycle t+1. Nevertheless, these items are not critical. Basically, during the instruction-address generation process, the most time consuming actions are the accesses to the BTB and the PT. Therefore, pipelining the instruction address generation process as described above allows the use of a shorter cycle than with conventional address generators and/or to implement bigger prediction tables without introducing any one-cycle penalty in the presence of a predicted taken branch.

Misprediction penalty Using a two-block ahead branch predictor in an out-of-order single I-fetch processor like the Intel PentiumPro or the MIPS R10000 does not result in a one-cycle increase of the misprediction penalty. As a matter of fact, the address of the non-predicted path is recorded in the checkpoint established for the branch to resume fetching in processors featuring a one-block ahead branch predictor. The two-block ahead scheme only requires to record in addition the non-predicted path in the IF2 stage (**AaT** or **AaN** if branch **Aa** was mispredicted) and the prediction made **Pa**, since all the other information required in stage IF2 to compute **Ci** would be known (fall-through address and return stack values).

In in-order single I-fetch processors, a structure can hold such values. Otherwise, the misprediction penalty would be increased by one cycle. This extra cycle is required to retrieve both the prediction and the possible addresses of the target instruction **Ci**. Nevertheless, one should note that replacing an instruction address generator resulting in pipeline bubbles on branches as in the DEC 21164 by our predictor would save the bubble in all cycles where the prediction is correct. For incorrect predictions, the penalty is the same for both mechanisms.



(a) Impact of the size of the PT

(b) Individual Results with a 64k-entry PT

Figure 9: Branch Misprediction Rates.

6 Experimental Results

Trace-driven simulations were conducted to verify the effectiveness of the two-block ahead branch predictor. We first establish that the branch prediction accuracy achieved by our branch prediction table is equivalent to those obtained with conventional one-block ahead branch prediction tables. Finally, we show that the two-block ahead branch predictor does not require any additional BTB logic to handle most branches.

6.1 Branch Prediction Accuracy

We assumed a perfect BTB (all branches hit) in these simulations to compare predictors without any clouding effects from the BTB. The simulations were run only over the CINT92 suite since floating-point benchmarks tend to lower misprediction rates.

Figure 9 (a) presents the average misprediction rate for two common prediction schemes with respect to the size of the prediction table. The misprediction rates are reported for both the two-block ahead branch predictor (*g-share 2*, *g-select 2*) and the corresponding one-block ahead branch predictor (*g-share*, *g-select*). These branch prediction schemes differ by the index which is used to access the prediction table. The prediction table in *g-select* is indexed with a concatenation of branch history and branch address bits. The index value in *g-share* is the exclusive OR of the branch address with the branch history register.

Notice that, for both schemes and for all table sizes, the performance of the two-block ahead branch predictors is very close to the performance of the corresponding one-block ahead branch predictors. The difference between the misprediction rates of the different benchmarks are reported in figure 9 (b) for a 64 K-entry prediction table. These differences are very tiny and do not exceed 0.30 %.

From these simulation results, we conclude that the two-block ahead branch history register and the two-

block ahead address are as representative of a branch as the conventional branch history register and the branch address.

6.2 The Branch Target Buffer

In the previous sections, we have introduced the two-block ahead branch target buffer to predict two block addresses per cycle or to pipeline the address generation process. Here our results verify that such a mechanism does not require a high degree of associativity or a large number of entries since a branch can be associated with more than one BTB entry.

Figure 10 reports the individual results with a 512-entry and a 2K-entry BTB with varying degrees of associativity. A pseudo-random replacement policy was used and the cache line size was assumed to be 16 instructions wide as in the MIPS R10000 and most of the current out-of-order processors. All the branches in the same cache line map to the same set in the BTB. In addition in the two-block ahead BTB, different types of branches may be associated with the same address tag (**AaT** and **AaN** for instance). Thus a set-associative BTB is required.

We can see from figures 10 (a) and 10 (b) that the maximum hit rate is nearly reached with an associativity of 4, compared to an associativity of 2 for a conventional BTB (figures 10 (c) and 10 (d)). These results also show that for realistic BTB sizes, the hit rate for a conventional BTB is slightly better than that for a two-block ahead BTB. However, this difference is less than 0.5 % for most applications (including gcc), so the improvement of fetching two blocks per cycle is still valuable. Thus, the two-block ahead branch predictor does not require any additional storage in the BTB, nor does it lead to any increase of the associativity.

7 Summary and Concluding Remarks

The current instruction-fetch mechanisms limit the performance that may be achieved. New solutions must be implemented in next generation microprocessors.

Two design philosophies have been used to achieve higher performance for the past four years. "Brainiac" processors attempt to achieve the highest level of IPC possible. Future generation "brainiac" processors should fetch more than one basic block in a single cycle, otherwise the fetch limit of one basic block per cycle would significantly impair performance. This raises the difficult issue of predicting multiple instruction blocks in parallel. On the other hand, "speed demon" processors get high-performance by increasing the clock rate. All parts of the processor must be pipelined and some functions are spread over several cycles (e.g. I-cache access). However the address-generation process is not pipelined in current designs. In these processors, either the address-generation mechanism (including branch prediction) becomes the electrical critical path or pipeline bubbles are inserted for each predicted taken branch. As this may severly limit the performance achieved in future designs, pipelining the address generation and the branch prediction is also a major issue.

We have introduced the Two-Block Ahead Branch Predictor to deal with both issues. In conventional branch prediction mechanisms, information associated with the current instruction block such as the address of the branch instruction is used to predict the next instruction block. The two-block ahead branch predictor

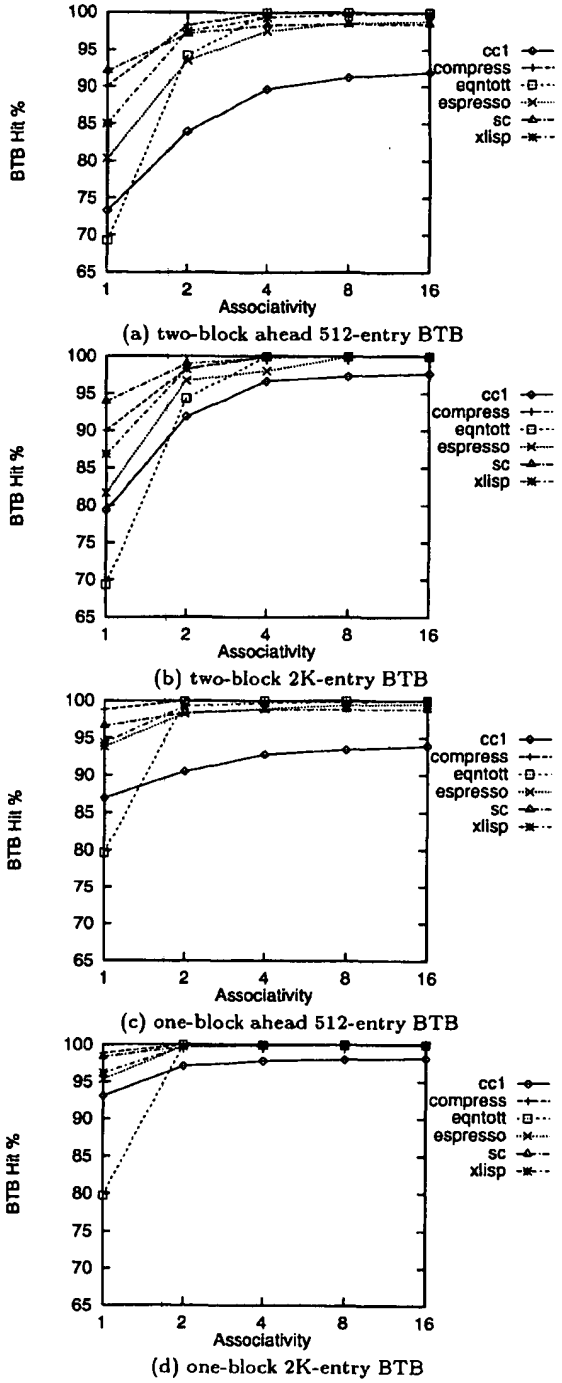


Figure 10: BTB Hit Rates According to the Associativity and the Number of Entries.

uses the same information when predicting the block following the next instruction block. The amount of information stored in our predictor is in the same range as in a conventional branch prediction mechanism. Furthermore, any branch prediction schemes proposed for single I-fetch processors can be adapted to our predictor. Simulations have shown that equivalent branch prediction accuracy is achieved. Thus, the two-block ahead branch predictor can be used to predict the address of two basic blocks in a single cycle, improving the hardware ILP of "brainiac" processors. It can also be used to pipeline the address-generation process over two cycles in "speed-demon" processors. The prediction accuracy remains the same.

The two-block ahead branch predictor can be extended to a multiple-block ahead branch predictor fetching multiple basic blocks in a single cycle or to further pipeline the address generation process over more than two cycles. We plan to study how the scheme scales from two-block to multiple-block ahead, especially on the return address structures, and on the accuracy and the features of the prediction structures. Any combination between the multiple-block ahead branch predictor and the previously proposed multiple schemes [19, 4] may worth be investigated to implement a high-end fetch mechanism or to pipeline a double I-fetch processor for instance.

The structures of the two-block ahead BTB and the two-block ahead PT presented in this paper have been directly deduced from existing conventional one-block ahead solutions. We are now investigating specific implementations of those two-block ahead structures. For instance, the double I-fetch implementation of our predictor features two dual-ported memory structures. We are looking at ways to build fast and cost effective structures by taking into account the correlation between basic blocks. We are also looking at ways to adapt cost-effective solutions for one-block ahead BTBs [2, 15] to two-block ahead BTBs.

Acknowledgments

We gratefully acknowledge the help and encouragement of the members of the HPS research group at ACAL, University of Michigan, during our work on this project, in particular, Eric Hao, Sanjay Patel, Daniel Friendly, Paul Racunas, Lee Hwang Lee, Tse Hao Hsing, Darren Vengroff, and Professor Yale Patt. They gave many inputs and provided critical comments on early versions of the paper. We also thank Richard Uhlig, presently at IRISA, for his help in polishing the final version, and Andy Glew (Intel) and the anonymous reviewers for their insightful comments.

References

[1] **M. Butler and Y. N. Patt,** "An Investigation of the Performance of Various Dynamic Scheduling Techniques," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[2] **B. Calder and D. Grunwald,** "Next Cache Line and Set Prediction," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[3] **T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel,** "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[4] **S. Dutta and M. Franklin,** "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[5] **L. Gwennap,** "Digital Leads the Pack with 21164," *Microprocessor Report*, September 1994.

[6] **L. Gwennap,** "Comparing RISC Microprocessors," *Proceedings of the Microprocessor Forum*, October 1994.

[7] **L. Gwennap,** "PA-8000 Combines Complexity and Speed," *Microprocessor Report*, November 1994.

[8] **L. Gwennap,** "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, February 1995.

[9] **IBM and Motorola,** "PowerPC 604 RISC Microprocessor User's Manual," MPR604UMU-01, 1994.

[10] **M. Johnson,** Superscalar Microprocessor Design, *Prentice-Hall*, 1991.

[11] **S. Jourdan, P. Sainrat, and D. Litaize,** "An Investigation of the Performance of Various Instruction-Issue Buffer Topologies" *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[12] **S. McFarling,** "Combining Branch Predictors," *Technical Note TN-36*, DEC-WRL, June 1993.

[13] **Mips Technologies Incorporated,** "R10000 Microprocessor Product Overview," *Technical Report*, October 1994.

[14] **D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi,** "Control Flow Prediction for Dynamic ILP Processors," *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[15] **A. Seznec,** "Don't use the page number, but a pointer to it," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[16] **M. D. Smith,** "Tracing with Pixie," *Technical report*, Stanford University, April 1991.

[17] **SPEC 92,** *Technical report*, December 1992.

[18] **S. Weiss and J. E. Smith,** POWER and PowerPC: Principles, Architecture and Implementation, *Morgan Kaufmann Publishers Inc.*, 1994.

[19] **T. Yeh, D. T. Marr, and Y. N. Patt,** "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[20] **T. Yeh,** "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," *PhD thesis*, Department of Electrical Engineering and Computer Science, University of Michigan, 1993.